Legal Information

OLE Programmer's Reference

Overview

Today's software applications are more capable and easier to use than ever before. Yet as their feature sets have grown in complexity and size, they have become increasingly difficult and costly to engineer, maintain, and upgrade. Adding new features always runs the risk of introducing new, sometimes intractable bugs and endangers backward compatibility with earlier versions. In addition, most applications are monolithic, providing rich sets of features but no easy way to add missing features or remove unneeded ones. Applications that do support adding and removing features in the form of plug-in components may expose their services to sibling applications from the same vendor but rarely to those from entirely different vendors. As a result, applications from different vendors typically work together poorly or not at all.

Operating systems have a related set of problems. Because most operating systems are not sufficiently modular, upgrading, replacing, or overriding existing services in a clean, flexible way is difficult. Like application developers, systems vendors face the difficult choice of foregoing or compromising needed improvements in order to maintain backward compatibility, or disenfranchising existing users in order to move the technology forward. These are choices that serve no one.

Software developers have recognized for some time that object-oriented programming offers promising solutions to several of these problems. By encapsulating data and functions in a single entity, then providing access to this rich content through a single reference, or pointer, object-oriented programming makes it possible to break down monolithic applications into functional modules, or *components,* that can be added or removed as they are needed. Moreover, object-oriented programming can potentially reduce the expense of implementing new objects by enabling them, through mechanisms such as inheritance, polymorphism, and aggregation, to make use of the capabilities built into already-existing objects.

Even so, object-oriented programming has yet to reach its full potential because no standard framework has existed through which software objects created by different vendors could interact with one another within the same address space, much less across process, machine, and network boundaries. Once you have broken down software into manageable, interchangeable components, you need some way for those components to communicate. That's where OLE comes in.

OLE provides a standard conceptual framework for creating, managing, and accessing object-based components that provide services to other objects and applications. OLE components can exist as parts of an operating system or application, or as stand-alone entities, and the services they provide can be most anything that operating systems and applications currently supply. You can use OLE components to expose both data and services to programs created by other vendors and write your own applications in such a way as to take advantage of services provided by others. You can also extend or customize basic OLE services to suit the needs of your objects and applications while guaranteeing that they continue to work with other OLE components.

Explaining the Component Object Model

The bedrock on which OLE rests is the Component Object Model (COM), which provides both the programming model and binary standard for OLE components. COM defines and implements mechanisms that enable software components, such as applications, data objects, controls, and services, to interact as *objects*. A software object comprises some body of data, along with one or more functions for accessing and using that data. An OLE component is one in which access to an object's data is achieved exclusively through one or more sets of related functions called *interfaces.*

An object provider, or *server*, makes an OLE component available by implementing one or more interfaces. A prospective user, or *client,* of an OLE component gains access to the object by obtaining a pointer to one of its interfaces. With this pointer, the client can reliably use the object without understanding its implementation and with the assurances that the object will always behave the same way. In this sense, an object's interface is a contract defining its behavior for prospective clients. The object will honor that contract even if its client lives in a different process or on a different machine, runs under a different operating system, is written by a different software developer using a different computer language, or represents an earlier or later version than the client used before.

By defining interfaces as contracts between objects and their clients, COM effectively solves the versioning problem. Here's how. To create a new version of an object, you simply add new interfaces while leaving the older ones in place. You define the new interfaces in such a way that clients are permitted to query for either the new version of the interface, or the old version, but not for both. Because of this restriction, adding new interfaces, or new versions of older interfaces, in no way interferes with the way existing clients work with an object.

Because OLE components conform to a binary standard, you can implement them using nearly any programming language. Object-oriented languages such as C++ and Smalltalk are ideal for this purpose, but any procedural language, such as C or Pascal, that supports the notion of a reference, or pointer, to an object will do. OLE components implemented in one programming language will work, without modification, with applications or other objects implemented in another language. In addition, because OLE components are language-independent, they are not constrained by language-based requirements to operate in the same address space as the objects or applications using them. As a result, OLE components provide the basis for sharing information among separate objects in different processes on the same machine or on different networked computers.

OLE Information Management

Built on top of COM are core facilities for storing and naming objects and for transferring data between them. Together, these facilities form the core of system-wide information management. You can take advantage of these facilities by using OLE's supplied implementations, or you can extend or replace them with your own. These core facilities include structured storage, monikers, and uniform data transfer. Together they provide the infrastructure required for objects to interrelate across process, machine, and network boundaries.

OLE's structured storage model defines an architecture for storing and retrieving objects that reside inside files or other containers. The basic problem addressed by the structured storage model is how to write an object to and retrieve an object from any persistent storage when that object resides inside the flat file space of some container. OLE solves the problem by providing interfaces that support *storage* and *stream* objects. A storage object provides a hierarchical structure for mapping the location of storage and stream objects just as directories provide a way of locating files on disk. A stream object contains the object's data, or contents. Using structured storage is mandatory if you want to support compound documents or other forms of containment. OLE also provides a default implementation of structured storage called *compound files.*

OLE supports the persistent naming of objects through a mechanism called a *moniker.* A moniker provides a way for clients of an object to locate that object even when it resides in a different process or on a different computer. Objects have a file moniker, which is similar to their absolute pathnames, but their clients normally bind to them using a relative moniker, which indicates the object's location relative to the client. In this way, if client and object are both relocated, they can maintain their connection as long as their locations relative to each other remain the same. In general, a moniker also enables clients to locate and connect to some portion of an object – a certain range of cells in a spreadsheet, for example – by providing an *item moniker,* by which that *pseudo-object* is always known.

OLE's uniform data transfer model provides a single, standard mechanism for transferring objects and data between applications. Through a single interface, OLE supports standard Microsoft® Windows® clipboard transfers, as well as drag and drop, and embedded objects. Uniform data transfer greatly enhances the clipboard model by providing rich structures for describing the format of the data to be transferred and the storage medium in which a transfer is to occur. For example, instead of always having to pass data in a global memory handle, you can choose from among several storage media, including a disk-based file and pointers to OLE storage and stream interfaces.

Built on top of COM, structured storage, monikers, and uniform data transfer are the three technologies for which OLE is best known: compound documents, Automation, and OLE controls. For many software vendors, the desire to incorporate one or both of these technologies in their applications, or to make sure their applications work easily with operating systems and other applications that do, has been the prime motivation for turning to OLE.

Describing Compound Documents

A *compound document* is one that contains, along with its native data, one or more objects that were created in other applications and therefore have different data formats. Such objects are called *compound document objects.* An application used to create compound document objects is called an *OLE server application.* An application whose documents act as object clients is called a *container application* or, simply, a *container*. Any given application can be one, or the other, or both.

Compound document objects are essentially OLE components that also implement interfaces that support object linking and embedding. A linked object includes information about the location of its data and the formats necessary to present it on screen to the user, but the data itself is stored elsewhere. An embedded object is one whose data is stored along with that of its container. Applications that support linking and embedding can share data without having knowledge of one another's data structures and without having to implement separate protocols for every other application with which they communicate. Users of OLE compound document applications can embed data created in one application in a document created in another, and can view, edit, or otherwise manipulate that data without having to exit the application in which they are working. Users can also create links in one document to data in another so that changes to data in the source document are updated in the link.

Defining Automation

Automation is the ability for one application or tool to manipulate programmatically objects exposed by another application. Using Automation, you can also create tools that access and manipulate objects. Such tools can include embedded macro languages, programming tools, object browsers, and compilers. Like compound documents, Automation is based on COM, but applications can implement Automation independently of compound documents or other OLE technologies.

Where to Find Additional Information

The current release of *OLE Programmer's Guide* examines the architecture and capabilities of OLE as a whole, as well as of each of its core facilities: COM, structured storage, and uniform data transfer. A separate chapter is devoted to compound documents, the technology that inspired the original development of OLE and laid the groundwork for its evolution into a systemwide, object-based service architecture. A subsequent release of the *OLE Programmer's Guide* will include detailed instructions for developing both OLE client and server applications for compound documents that work under both Microsoft® Windows NT® and Microsoft® Windows® 95. For descriptions of specific interfaces, helper functions, data structures, and enumerations, refer to the *OLE Programmer's Reference.* For information on Automation and developing OLE controls, see the *Automation Programmer's Guide and Reference*.

The Component Object Model

The Component Object Model (COM) is the base technology of OLE, a broad set of object-oriented technology standards. OLE includes, besides COM, object design standards at a higher level. Among these are standards for OLE Structured Storage, OLE Compound Documents, and OLE Controls. COM is the binary standard that defines the means for applications to interact within these technology standards.

To understand COM (and therefore all OLE technologies), it is crucial to bear in mind that it is not an object-oriented language, but a standard. Nor does COM specify how an application should be structured. Language, structure, and implementation details are left to the application programmer. COM does specify an object model and programming requirements that enable COM objects (also called OLE Components, or sometimes simply *objects*) to interact with other objects. These objects can be within a single process, in other processes, even on remote machines. They can have been written in other languages, and may be structurally quite dissimilar. That is why COM is referred to as a binary standard – it is a standard that applies after a program has been translated to binary machine code.

The only language requirement for COM is that code is generated in a language that can create structures of pointers and, either explicitly or implicitly, call functions through pointers. Object-oriented languages such as C++ and Smalltalk provide programming mechanisms that simplify the implementation of COM objects, but languages such as C, Pascal, Ada, Java, and even BASIC programming environments can create and use COM objects.

COM defines the essential nature of a COM object. In general, a software object is made up of a set of data and the functions that manipulate the data. A COM object is one in which access to an object's data is achieved exclusively through one or more sets of related functions. These function sets are called *interfaces*, and the functions of an interface are called *methods*. Further, COM requires that the only way to gain access to the methods of an interface is through a pointer to the interface.

Besides specifying the basic binary object standard, COM defines certain basic interfaces that provide functions common to all COM-based technologies. It also provides a small number of API functions that all components require. COM has now expanded its scope to define how objects work together over a distributed environment, and added security features to ensure system and component integrity.

This chapter describes basic COM issues relating mainly to designing COM objects:

- **COM Objects and Interfaces**
- **Using and Implementing IUnknown**
- **Reusing Objects**
- **The COM Library**
- **Managing Memory Allocation**
- **Processes and Threads**

For other COM topics, see **COM Clients and Servers**.

COM Objects and Interfaces

COM is a technology that allows objects to interact across process and machine boundaries as easily as objects within a single process interact. COM enables this by specifying that the only way to manipulate the data associated with an object is through what is called an *interface on the object.* When this term is used, it refers to an implementation in code of a COM binary-compliant interface that is associated with an object.

Talking about an object that *implements an interface* means that the object uses code that implements each method of the interface and provides COM binary-compliant pointers to those functions to the COM library. COM then makes those functions available to any client who asks for a pointer to the interface, whether the client is inside or outside of the process that implements those functions.

For more information, see the following:

- **Interfaces and Interface Implementations**
- **Interface Pointers and Interfaces**
- **IUnknown and Interface Definition Inheritance**

Interfaces and Interface Implementations

COM makes a fundamental distinction between interface definitions and their implementations. An *interface* is actually a contract that consists of a group of related function *prototypes* whose usage is defined but whose implementation is not. These function prototypes are equivalent to pure virtual base classes in C++ programming. An interface definition specifies the interface's member functions, called *methods,* their return types, the number and types of their parameters, and what they must do. There is no implementation associated with an interface.

An *interface implementation* is the code a programmer supplies to carry out the actions specified in an interface definition. Implementations of many of the interfaces a programmer could use in an object-based application are included in the OLE libraries. Programmers are, however, free to ignore these implementations and write their own. An interface implementation is to be associated with an object when an instance of that object is created, and provides the services that the object offers.

For example, a hypothetical interface named **IStack** (many interface names begin with the letter "I") might define two *methods,* named **Push** and **Pop**, specifying that successive calls to the **Pop** method return, in reverse order, values previously passed to the **Push** method. This interface definition, however, would not specify how the functions are to be implemented in code. In implementing the interface, however, one programmer might implement the stack as an array and implement the **Push** and **Pop** methods in such a way as to access that array; while another programmer might prefer to use a linked list and would implement the methods accordingly. Regardless of a particular implementation of the **Push** and **Pop** methods, however, the in-memory representation of a pointer to an **IStack** interface, and therefore its use by a client, is completely defined by the interface definition.

Simple objects may support only a single interface. More complicated objects, such as embeddable objects, typically support several interfaces. Clients have access to an OLE object only through a pointer to one of its interfaces, which, in turn, allows the client to call any of the methods that make up that interface. These methods determine how a client can use the object's data.

Interfaces, in fact, define a contract between an object and its clients. The contract specifies the methods that must be associated with each interface, and what the behavior of each of the methods must be in terms of input and output. The contract generally does not define *how* to implement the methods in an interface. Another important aspect of the contract is that if an object supports an interface, it must support all of its methods in some way. Not all of the methods in an implementation need to do something − if an object does not support the function implied by a method, its implementation may be a simple return, or perhaps the return of a meaningful error message − but the methods must *exist*.

COM uses the word "interface" in a sense different from that typically used in C++ programming. A C++ interface refers to *all* of the functions that a class supports and that clients of an object can call to interact with it. A COM interface refers to a predefined group of related functions that a COM class implements, but does not necessarily represent *all* the functions that the class supports.

Interface Pointers and Interfaces

An instance of an interface implementation is actually a pointer to an array of pointers to methods (a function table that refers to an implementation of all of the methods specified in the interface). Objects with multiple interfaces can provide pointers to more than one function table. Any code that has a pointer through which it can access the array can call the methods in that interface.

Speaking precisely about this multiple indirection is inconvenient, so instead, the pointer to the interface function table that another object must have to call its methods is called simply an *interface pointer*. You can manually create function tables in a C application or almost automatically with C++ (or other object-oriented languages that support COM).

With appropriate compiler support (which is inherent in C and C++), a client can call an interface method through its name, not its position in the array. Because an interface is a type, given the names of methods the compiler can check the types of parameters and return values of each interface method call. In contrast, such type-checking is not available even in C or C++ if a client uses a position-based calling scheme.

Each interface – the immutable contract of a functional group of methods – is referred to at runtime with a globally-unique interface identifier, an "IID". This IID, which is a specific instance of a GUID (a Globally Unique IDentifier supported by OLE) allows a client to ask an object precisely if it supports the semantics of the interface without unnecessary overhead and without the confusion that could arise in a system from having multiple versions of the same interface with the same name.

To summarize, it is important to understand what an interface is, and is not:

- An interface is not the same as a C++ class.
- An interface is not an object.
- Interfaces are strongly typed.
- Interfaces are immutable.

An interface is not a C++ class–the pure virtual definition carries no implementation. If you are a C++ programmer, you can, however, define your implementation of an interface as a class, but this falls under the heading of implementation details, which COM does not specify. An instance of an object that implements an interface must be created for the interface actually to exist. Furthermore, different object classes may implement an interface differently yet be used interchangeably in binary form, as long as the behavior conforms to the interface definition.

An interface is not an object – it is simply a related group of functions and is the binary standard through which clients and objects communicate. The object can be implemented in any language with any internal state representation, as long as it can provide pointers to interface methods.

Interfaces are strongly typed – every interface has its own interface identifier (a GUID), which eliminates the possibility of duplication that could occur with any other naming scheme.

Interfaces are immutable contracts – you cannot define a new version of an old interface and give it the same identifier. Adding or removing methods of an interface, or changing semantics creates a new interface, not a new version of an old interface. Therefore a new interface cannot conflict with an old interface. Objects can, of course, support multiple interfaces simultaneously, and can expose interfaces that are successive revisions of an interface, with different identifiers. Thus, each interface is a separate contract, so system-wide objects need not be concerned about whether the version of the interface they are calling is the one they expect. The interface ID (IID) defines the interface contract explicitly and uniquely.

IUnknown and Interface Definition Inheritance

Inheritance in COM does not mean code reuse. Because no implementations are associated with interfaces, interface inheritance does not mean code inheritance. It means only that the contract associated with an interface is inherited in a C++ pure-virtual base-class fashion and modified – either by adding new methods or by further qualifying the allowed usage of methods. There is no selective inheritance in COM: If one interface inherits from another, it includes all the methods that the other interface defines.

Inheritance is used sparingly in the predefined COM interfaces. All predefined interfaces (and any custom interfaces you define) inherit their definitions from the important interface **IUnknown**, which contains three vital methods: **QueryInterface**, **AddRef**, and **Release**. All COM objects must implement the **IUnknown** interface, because it provides the means to move freely between the different interfaces that an object supports with **QueryInterface**, and to manage its lifetime with **AddRef** and **Release**. More on these methods is discussed later in this chapter.

Any single object usually requires only a single implementation of the **IUnknown** methods. This means that by implementing any interface on an object you must completely implement the **IUnknown** functions. You do not generally need to explicitly inherit from nor implement **IUnknown** as its own interface. When a client of the object queries for it, it is usually necessary only to typecast another interface pointer into an **IUnknown**, as this interface makes up the first three entries in the function table already.

In some specific situations, notably in creating an object that supports aggregation, you may need to implement one set of **IUnknown** functions for all interfaces as well as a stand-alone **IUnknown** interface. This is described in the following section. In any case, any object implementor will implement **IUnknown** methods. Refer to the section **Using and Implementing IUnknown** for more information.

While there are a few interfaces that inherit their definitions from a second interface, in addition to **IUnknown**, the majority are simply the **IUnknown** interface methods plus the methods defined in the interface. This makes most interfaces relatively compact and easy to encapsulate.

Using and Implementing IUnknown

COM provides a rich set of standards for implementing and using objects, and inter-object communication. The following sections describe basic information relating to implementing objects. For information on how clients and servers interact, refer to **COM Clients and Servers**. For specific information on threading models, their implementation and use, refer to **Processes and Threads**.

For details on using and implementing **IUnknown**, see the following:

- **QueryInterface: Navigating in an Object**
- **Rules for Implementing QueryInterface**
- **Managing Object Lifetimes through Reference Counting**

QueryInterface: Navigating in an Object

Once you have an initial pointer to an interface on an object, OLE has a very simple mechanism to find out whether the object supports another specific interface, and, if so, to get a pointer to it. (For information on getting an initial pointer to an interface on an object, refer to Getting a Pointer to an Object.) This mechanism is the **QueryInterface** method of the **IUnknown** interface. If the object supports the requested interface, the method must return a pointer to that interface. This permits an object to navigate freely through the interfaces that an object supports. **QueryInterface** separates the request "Do you support a given contract?" from the high-performance use of that contract once negotiations have been successful.

When a client initially gains access to an object, that client will receive, at a minimum, an **IUnknown** interface pointer (the most fundamental interface) through which it can control the lifetime of the object – tell the object when it is done using the object – and invoke **QueryInterface**. The client is programmed to ask each object it manages to perform some operations, but the **IUnknown** interface has no functions for those operations. Instead, those operations are expressed through other interfaces. The client is thus programmed to negotiate with objects for those interfaces. Specifically, the client will ask an object – by calling **QueryInterface** – for an interface through which the client may invoke the desired operations.

Since the object implements **QueryInterface**, it has the ability to accept or reject the request. If the object accepts the client's request, **QueryInterface** returns a new pointer to the requested interface to the client. Through that interface pointer, the client has access to the methods of that interface. If, on the other hand, the object rejects the client's request, **QueryInterface** returns a null pointer – an error – and the client has no pointer through which to call the desired functions. In this case, the client must deal gracefully with that possibility. For example, suppose a client has a pointer to interface A on an object, and asks for interfaces B and C. While the object supports interface B, it does not support interface C. The object returns a pointer to B, and reports that C is not supported.

A key point is that when an object rejects a call to **QueryInterface**, it is impossible for the client to ask the object to perform the operations expressed through the requested interface. A client *must* have an interface pointer to invoke methods in that interface. There is no alternative. If the object refuses to provide one, a client must be prepared to do without, simply not doing whatever it had intended to do with that object. Had the object supported that interface, the client might have done something useful with it. This works well in comparison with other object-oriented systems in which you cannot know whether a function will work until you call that function, and even then, handling failure is uncertain. **QueryInterface** provides a reliable and consistent way to know whether an object supports an interface before attempting to call its methods.

The **QueryInterface** method also provides a robust and reliable way for an object to indicate that it does *not* support a given contract. That is, if in a call to **QueryInterface** one asks an "old" object whether it supports a "new" interface (one, for example, that was invented after the old object had been shipped), the old object will reliably and, without causing a crash, answer "no." The technology that supports this is the algorithm by which IIDs are allocated. While this may seem like a small point, it is extremely important to the overall architecture of the system, and the ability to inquire of old things about new functionality is, surprisingly, a feature not present in most other object architectures.

Rules for Implementing QueryInterface

There are three main rules than govern implementing the **IUnknown::QueryInterface** method on a COM object:

- Objects must have identity
- The set of interfaces on an object must be static
- It must be possible to query successfully for any interface on an object from any other interface

Objects must have identity. For any given object instance, a call to **QueryInterface** must always return the same physical pointer value. This allows you to call **QueryInterface(IID_IUnknown, ...)** on any two interfaces and compare the results to determine whether they point to the same instance of an object.

The set of interfaces on an object must be static. The set of interfaces accessible on an object via **QueryInterface** must be static, not dynamic. Specifically, if **QueryInterface** returns S_OK for a given IID once, it must never return E_NOINTERFACE on subsequent calls on the same object, and if **QueryInterface** returns E_NOINTERFACE for a given IID, subsequent calls for the same IID on the same object must never return S_OK.

It must be possible to query successfully for any interface on an object from any other interface. That is, given the following code:

```
IA * pA = (some function returning an IA *);
IB * pB = NULL;
HRESULT   hr;
hr = pA->QueryInterface(IID_IB, &pB); // line 4
```

the following rules apply:

- If you have a pointer to an interface on an object, a call like the following to **QueryInterface** for that same interface must succeed:

  ```
  pA->QueryInterface(IID_IA, ...)
  ```

- If a call to **QueryInterface** for a second interface pointer succeeds, a call to QueryInterface from that pointer for the first interface must also succeed. If in line 4 of the example, *pB* was successfully obtained, the following must also succeed:

  ```
  pB->QueryInterface(IID_IA, ...)
  ```

- Any interface must be able to query for any other interface on an object. So in line 4 of the example, *pB* was successfully obtained, and you successfully query for a third interface (IC) using that pointer, you must also be able to query successfully for IC using the first pointer, pA. In this case, the following sequence must succeed:

  ```
  IC * pC = NULL;
  hr = pB->QueryInterface(IID_IC, &pC);      //Line 7
  pA->QueryInterface(IID_IC, ...)
  ```

Interface implementations must maintain a counter that is large enough to support 2_{31}-1 outstanding pointer references to all the interfaces on a given object. Therefore, you should use a 32-bit unsigned integer for the counter.

If a client needs to know that resources have been freed, it must use a method in some interface on the object with higher-level semantics before calling **Release**.

Managing Object Lifetimes through Reference Counting

In traditional object systems, the life cycle of objects – the issues surrounding the creation and deletion of objects – is handled implicitly by the language (or the language runtime) – instances go explicitly out of scope as parameters or global variables, for example – or explicitly by application programmers.

In an evolving, decentrally constructed *system* made up of reused components, it is no longer true that any one or client, or even programmer, always "knows" how to deal with a component's lifetime. For a client with the right security privileges, it is still relatively easy to create objects through a simple request. But object deletion is another matter entirely. It is not necessarily clear when an object is no longer needed and should be deleted. Even when the original client is done with the object, it cannot simply shut the object down, because some other client (or clients) in the system may still have a reference to it.

One way to ensure that an object is no longer needed, so it can be deleted, is to depend entirely upon an underlying communication channel to inform the system when all connections to a cross-process or cross-channel object have disappeared. Schemes that use this method are unacceptable for several reasons. One problem is that it could require a major difference between the cross-process/cross-network programming model and the single-process programming model. In the cross-process/cross-network programming model, the communication system would provide the hooks necessary for object lifetime management, while in the single-process programming model, objects are directly connected without any intervening communications channel. Another problem is that this scheme could also result in a layer of system-provided software that would interfere with component performance in the in-process case. Furthermore, such a mechanism based on explicit monitoring would not tend to scale towards many thousands or millions of objects.

COM offers a scalable and distributed approach to this set of problems. Clients tell an object when they are using it and when they are done, and objects delete themselves when they are no longer needed. This approach mandates that all objects count references to themselves.

Just as an application must free memory it has allocated once that memory is no longer in use, a client of an object is responsible for freeing its references to the object when that object is no longer needed. In an object-oriented system, the client can only do this by giving the object an instruction to free itself.

It is important that an object be deallocated when it is no longer being used. The difficulty lies in determining when it is safe to deallocate an object. This is easy with automatic variables (those allocated on the stack) – they cannot be used outside the block in which they're declared, so the compiler deallocates them when the end of the block is reached. For COM objects, which are dynamically allocated, it is up to the clients of an object to decide when they no longer need to use the object – especially local or remote objects that may be in use by multiple clients at the same time. The object must wait until *all* clients are finished with it before freeing itself. Because COM objects are manipulated through interface pointers and can be used by objects in different processes or on other machines, the system cannot keep track of an object's clients.

COM's method of determining when it is safe to deallocate an object is manual reference counting. Each object maintains a 32-bit reference count that tracks how many clients are connected to it, that is, how many pointers exist to any of its interfaces in any client.

Implementing Reference Counting

Reference counting requires work on the part of both the implementor of a class and the clients who use objects of that class. When you implement a class, you must implement the **AddRef** and **Release** methods as part of the **IUnknown** interface. These two functions have simple implementations:

1. **AddRef** increments the object's internal reference count.
2. **Release** first decrements the object's internal reference count; then it checks whether the reference count has fallen to zero. If it has, that means no one is using the object any longer, so the **Release** function deallocates the object.

A common implementation approach for most objects is to have only one implementation of these functions (along with **QueryInterface**), which are shared between all interfaces, and therefore a reference count which applies to the entire object. Architecturally, however, from a client's perspective, reference counting is strictly and clearly a *per-interface-pointer* notion, and objects may be implemented which take advantage of this capability by dynamically constructing, destroying, loading, or unloading portions of their functionality based on the currently extant interface pointers

Whenever a client calls a method (or API function) that returns a new interface pointer, such as **QueryInterface**, the method being called is responsible for incrementing the reference count through the returned pointer. For example, when a client first creates an object, it receives an interface pointer to an object that, from the client's point of view, has a reference count of one. If the client then calls **AddRef** on the interface pointer, the reference count becomes two. The client must call **Release** twice on the interface pointer to drop all of its references to the object.

An illustration of how reference counts are strictly per-interface-pointer occurs when a client calls **QueryInterface** on the first pointer for either a new interface or the same interface. *In either of these cases* the client is required to call **Release** once for each pointer. COM does not require that an object return the same pointer when asked for the same interface multiple times. (The only exception to this is a query to **IUnknown**, which acts as an object's *identity* to COM.) This allows the object implementation to manage resources efficiently.

Rules for Managing Reference Counts

Using a reference count to manage an object's lifetime allows multiple clients to obtain and release access to a single object, without having to coordinate with each other in managing the object's lifetime. As long as the client objects conform to certain rules of use, the object, in a sense, provides this management. These rules relate to how to manage references between objects – COM does not specify internal implementations of objects, although these rules are a reasonable starting point for a policy within an object.

Conceptually, interface pointers can be thought of as living in pointer variables that include all internal computation state that holds an interface pointer. This would include variables in memory locations and in internal processor registers, and both programmer- and compiler-generated variables. Assignment to or initialization of a pointer variable involves creating a *new copy* of an already existing pointer. Where there was one copy of the pointer in some variable (the value used in the assignment/initialization), there are now two. An assignment to a pointer variable *destroys* the pointer copy presently in the variable, as does the destruction of the variable itself (that is, the scope in which the variable is found, such as the stack frame, is destroyed).

From a COM client's perspective reference counting is *always* done for each interface. Clients should never assume that an object uses the same counter for all interfaces.

The default case is that **AddRef** must be called for every new copy of an interface pointer, and **Release** called for every destruction of an interface pointer except where the following rules permit otherwise:

In-Out parameters to functions
: The caller must call **AddRef** on the parameter, since it will be released (with a call to **Release**) in the implementing code when the out-value is stored on top of it.

Fetching a global variable
: When creating a local copy of an interface pointer from an existing copy of the pointer in a global variable, you must call **Addref** on the local copy, because another function might destroy the copy in the global variable while the local copy is still valid.

New pointers synthesized out of "thin air."
: A function that synthesizes an interface pointer using special internal knowledge rather than obtaining it from some other source must call **AddRef** initially on the newly synthesized pointer. Important examples of such routines include instance creation routines, implementations of **IUnknown::QueryInterface**, etc.

Retrieving a copy of an internally stored pointer
: When a function retrieves a copy of a pointer that is stored internally by the object called, that object's code must call **AddRef** on the pointer before the function returns. Once the pointer has been retrieved, the originating object has no other way of determining how its lifetime relates to that of the internally stored copy of the pointer.

The only exceptions to this default case require that the managing code know the relationships of the lifetimes of two or more copies of a pointer to the same interface on an object, and simply ensure that the object is not destroyed by allowing its reference count to go to zero. There are generally two cases:

- When one copy of a pointer already exists, and a second is created subsequently, and then is destroyed while the first copy still exists, calls to **AddRef** and **Release** for the second copy can be omitted.
- When one copy of a pointer exists, and a second is created, and then the first is destroyed before the second, the calls to **AddRef** for the second copy and **Release** for the first copy can be omitted.

The following are specific examples of these situations, the first two being especially common:

In-parameters to functions
: The lifetime of the copy of an interface pointer passed as a parameter to a function is nested in that of

the pointer used to initialize the value, so there is no need for a separate reference count on the parameter,.

Out-parameters from functions, including return values
To set the out parameter, the function must have a stable copy of the interface pointer. On return, the caller is responsible for releasing the pointer. Thus, the out-parameter does not need a separate reference count.

Local variables
A method implementation has control of the lifetimes of each of the pointer variables allocated on the stack frame, and can use this to determine how to omit redundant **AddRef**/**Release** pairs.

Backpointers
Some data structures contain two objects each with a pointer to the other. If the lifetime of the first is known to contain the lifetime of the second, it is not necessary to have a reference count on the second's pointer to the first object. Often, avoiding this cycle is important in maintaining the appropriate deallocation behavior. However, uncounted pointers should be used with extreme caution, because the portion of the operating system that handles remote processing has no way of knowing about this relationship. Therefore, in almost all cases, having the backpointer refer to a second, "friend" object of the first pointer (thus avoiding the circularity) is the prefered solution. The Connectable Objects architecture, for example, uses this approach.

When implementing or using reference counted objects, it may be useful to use a technique called *artificial reference counts*, which guarantee object stability during processing of a function. In implementing a method of an interface, you may call functions that have a chance of decrementing your reference count to an object, causing a premature release of the object and failure of the implementation. A robust way to protect yourself from this is to insert a call to **AddRef** at the beginning of the method implementation, and pair it with a call to release **Release** just before the method returns.

The return values of **AddRef** and **Release** should not be relied upon and should be used only for debugging purposes.

Reusing Objects

An important goal of any object model is that object authors can reuse and extend objects provided by others as pieces of their own implementations. One way to do this in C++ and other languages is implementation inheritance, which allows an object to inherit ("subclass") some of its functions from another object while overriding other functions.

The problem for system-wide object interaction using traditional implementation inheritance is that the contract (the interface) between objects in an implementation hierarchy is not clearly defined. In fact, it is implicit and ambiguous. When the parent or child object changes its implementation, the behavior of related components may become undefined, or unstably implemented. In any single application, where the implementation can be managed by a single engineering team, who update all of the components at the same time, this is not always a major concern. In an environment where the components of one team are built through black-box reuse of other components built by other teams, this type of instability jeapordizes reuse. Additionally, implementation inheritance usually works only within process boundaries. This makes traditional implementation inheritance impractical for large, evolving systems composed of software components built by many engineering teams.

The key to building reusable components is to be able to treat the object as a black box. This means that the piece of code attempting to reuse another object knows nothing, and needs to know nothing, about the internal structure or implementation of the component being used. In other words, the code attempting to reuse a component depends upon the *behavior* of the object and not the exact *implementation*.

To achieve black-box reusability, COM adopts other established reusability mechanisms: *containment*/*delegation* and *aggregation*. For convenience in describing them, the object being reused is called the *inner object* and the object making use of that inner object is the *outer object*.

It is important to remember in both these mechanisms how the outer object appears to its clients. As far as the clients are concerned, both objects implement any interfaces to which the client can get a pointer. The client treats the outer object as a black box, and thus does not care, nor does it need to care, about the internal structure of the outer object – the client cares only about behavior.

For details on both of these mechanisms, see the following:

- **Containment/Delegation**
- **Aggregation**

Containment/Delegation

The first, and most common, mechanism is called containment/delegation. This type of reuse is a familiar concept found in most object-oriented languages and systems. The outer object acts as an object client to the inner object. The outer object "contains" the inner object and when the outer object requires the services of the inner object, the outer object explicitly delegates implementation to the inner object's methods. Thus, the outer object uses the inner object's services to implement itself.

It is not necessary for the outer and inner objects to support the same interfaces, although it certainly is reasonable to contain an object that implements an interface that the outer object does not, and implement the methods of the outer object simply as calls to the corresponding methods in the inner object. When the complexity of the outer and inner objects differs greatly, however, the outer object may implement some of the methods of its interfaces by delegating calls to interface methods implemented in the inner object.

It is simple to implement containment for an outer object. The outer object creates the inner objects it needs to use as any other client would. This is nothing new – the process is like a C++ object that itself contains a C++ string object that it uses to perform certain string functions, even if the outer object is not considered a "string" object in its own right. Then, using its pointer to the inner object, a call to a method in the outer object generates a call to an inner object method.

Aggregation

Aggregation is the other, richer, reuse mechanism, in which the outer object exposes interfaces from the inner object as if they were implemented on the outer object itself. This is useful when the outer object would always delegate every call to one of its interfaces to the same interface in the inner object. Aggregation is actually a specialized case of containment/delegation, and is available as a convenience to avoid extra implementation overhead in the outer object in these cases.

Aggregation is almost as simple to implement as containment is, except for the three **IUnknown** functions: **QueryInterface**, **AddRef**, and **Release**. The catch is that from the client's perspective, any **IUnknown** function on the outer object must affect the outer object. That is, **AddRef** and **Release** affect the outer object and **QueryInterface** exposes all the interfaces available on the outer object. However, if the outer object simply exposes an inner object's interface as its own, that inner object's **IUnknown** members called through that interface will behave differently than those **IUnknown** members on the outer object's interfaces, an absolute violation of the rules and properties governing **IUnknown**.

The solution is that aggregation requires an explicit implementation of **IUnknown** on the inner object and delegation of the **IUnknown** methods of any other interface to the outer object's **IUnknown** methods.

Creating Aggregable Objects

Creating objects that can be aggregated is optional; however, it is simple to do and to do so has significant benefits. The following rules apply to creating an aggregable object:

- The aggregable (or inner) object's implementation of **IUnknown::QueryInterface**, **AddRef**, and **Release** controls the inner object's reference count, and this implementation must not delegate to the outer object's unknown (the controlling **IUnknown**).
- The **QueryInterface**, **AddRef**, and **Release** methods of all other interfaces implemented on the inner object must delegate to the controlling **IUnknown** and not directly affect the inner object's reference count.
- The inner **IUnknown** must implement **QueryInterface** only for the inner object.
- The aggregable object must not call **AddRef** when holding a reference to the controlling **IUnknown** pointer.
- When the object is created, if any interface other than **IUnknown** is requested, the creation must fail with **E_UNKNOWN**.

The code fragment below illustrates a correct implementation of an aggregable object using the nested class method of implementing interfaces:

```
// CSomeObject is an aggregable object that implements
// IUnknown and ISomeInterface
class CSomeObject : public IUnknown
{
    private:
        DWORD        m_cRef;         // Object reference count
        IUnknown*    m_pUnkOuter;    // Controlling IUnknown, no AddRef

        // Nested class to implement the ISomeInterface interface
        class CImpSomeInterface : public ISomeInterface
        {
            friend class CSomeObject ;
            private:
                DWORD    m_cRef;    // Interface ref-count, for debugging
                IUnknown*    m_pUnkOuter;    // controlling IUnknown
            public:
```

```
                CImpSomeInterface() { m_cRef = 0;   };
                ~ CImpSomeInterface(void) {};

                // IUnknown members delegate to the outer unknown
                // IUnknown members do not control lifetime of object
                STDMETHODIMP     QueryInterface(REFIID riid, void** ppv)
                {    return m_pUnkOuter->QueryInterface(riid,ppv);   };

                STDMETHODIMP_(DWORD) AddRef(void)
                {    return m_pUnkOuter->AddRef();   };

                STDMETHODIMP_(DWORD) Release(void)
                {    return m_pUnkOuter->Release();   };

                // ISomeInterface members
                STDMETHODIMP SomeMethod(void)
                {    return S_OK;   };
        } ;
        CImpSomeInterface m_ImpSomeInterface ;
    public:
        CSomeObject(IUnknown * pUnkOuter)
        {
            m_cRef=0;
            // No AddRef necessary if non-NULL as we're aggregated.
            m_pUnkOuter=pUnkOuter;
            m_ImpSomeInterface.m_pUnkOuter=pUnkOuter;
        } ;
        ~CSomeObject(void) {} ;

        // Static member function for creating new instances (don't use
        // new directly).Protects against outer objects asking for
interfaces
        // other than IUnknown
        static HRESULT Create(IUnknown* pUnkOuter, REFIID riid, void **ppv)
        {
            CSomeObject*        pObj;
            if (pUnkOuter != NULL && riid != IID_IUnknown)
                return CLASS_E_NOAGGREGATION;
            pObj = new CSomeObject(pUnkOuter);
            if (pObj == NULL)
                return E_OUTOFMEMORY;
            // Set up the right unknown for delegation (the non-aggregation
case)
            if (pUnkOuter == NULL)
                pObj->m_pUnkOuter = (IUnknown*)pObj ;
            HRESULT hr;
            if (FAILED(hr = pObj->QueryInterface(riid, (void**)ppv)))
                delete pObj ;
            return hr;
        }

        // Inner IUnknown members, non-delegating
        // Inner QueryInterface only controls inner object
        STDMETHODIMP QueryInterface(REFIID riid, void** ppv)
        {
```

```
            *ppv=NULL;
            if (riid == IID_IUnknown)
                *ppv=this;
            if (riid == IID_ISomeInterface)
                *ppv=&m_ImpSomeInterface;
            if (NULL==*ppv)
                return ResultFromScode(E_NOINTERFACE);
            ((IUnknown*)*ppv)->AddRef();
            return NOERROR;
        } ;
        STDMETHODIMP_(DWORD) AddRef(void)
        {    return ++m_cRef; };
        STDMETHODIMP_(DWORD) Release(void)
        {
            if (--m_cRef != 0)
                return m_cRef;
            delete this;
            return 0;
        };
};
```

Aggregating Objects

When developing an aggregable object, the following rules apply:

- When creating the inner object, the outer object must explicitly ask for its **IUnknown**.
- The outer object must protect its implementation of **Release** from reentrancy with an artificial reference count around its destruction code.
- The outer object must call its controlling IUnknown's **Release** if it queries for a pointer to any of the inner object's interfaces. To free this pointer, the outer object calls its controlling IUnknown's **AddRef,** followed by **Release** on the inner object's pointer.

  ```
  // Obtaining inner object interface pointer
  pUnkInner->QueryInterface(IID_ISomeInterface, &pISomeInterface);
  pUnkOuter->Release();

  // Releasing inner object interface pointer
  pUnkOuter->AddRef();
  pISomeInterface->Release();
  ```

- The outer object must not blindly delegate a query for any unrecognized interface to the inner object, unless that behavior is specifically the intention of the outer object.

The COM Library

Any process that uses COM must both initialize and uninitialize the COM library. In addition to being a specification, COM also implements some important services in this library. Provided as a DLL in Microsoft® Windows®, the COM library includes:

- A small number of fundamental API functions that facilitate the creation of COM applications, both client and server. For clients, COM supplies basic functions for creating objects. For servers, COM supplies the means of exposing their objects.
- Implementation-locator services through which COM determines from a unique class identifier (CLSID) which server implements that class and where that server is located. This service includes support for a level of indirection, usually a system registry, between the identity of an object class and the packaging of the implementation such that clients are independent of the packaging, which can change in the future.
- Transparent remote procedure calls when an object is running in a local or remote server.
- A standard mechanism to allow an application to control how memory is allocated within its process, particularly memory that needs to be passed between cooperating objects such that it can be freed properly.

To use basic COM services, all COM threads of execution in clients and out-of-process servers must call either the **CoInitialize** or the **CoInitializeEx** function before calling any other COM function except memory allocation calls. **CoInitializeEx** replaces the other function, adding a parameter that allows you to specify the threading model of the thread – either apartment-threaded or free-threaded. A call to **CoInitialize** simply sets the threading model to apartment-threaded. OLE Compound Document applications call the **OleInitialize** function, which calls **CoInitialize** and also does some initialization required for compound documents. Since this is true, threads that call **OleInitialize** cannot be free-threaded. For information on threading in clients and servers, refer to **Processes and Threads**.

In-process servers do not call the initialization functions, because they are being loaded into a process that has already done so. As a result, in-process servers must set their threading model in the registry under the **InprocServer32** key. For detailed information on threading issues in in-process servers, refer to **In-Process Server Threading Issues**.

It is also important to uninitialize the library. For each call to **CoInitialize** or **CoInitializeEx**, there must be a corresponding call to **CoUninitialize**. For each call to **OleInitialize**, there must be a corresponding call to **OleUninitialize**.

In-process servers can assume that the process they are being loaded into has already performed these steps.

Managing Memory Allocation

In COM, many, if not most, interface methods and APIs are called by code written by one programming organization and implemented by code written by another. Many of the parameters and return values of these functions are of types that can be passed around by value. Sometimes, however, it is necessary to pass data structures for which this is not the case, so it is necessary for both caller and called to have a compatible allocation and de-allocation policy. COM defines a universal convention for memory allocation, both because it is more reasonable than defining case-by-case rules, and so the COM remote procedure call implementation can correctly manage memory.

The methods of COM interface always provide memory management of pointers to the interface by calling the **AddRef** and **Release** functions found in the **IUnknown** interface, from which all other COM interfaces derive (refer to Rules for Managing Reference Counts for more information). This section describes only how to allocate memory for parameters that are not passed by value – *not* pointers to interfaces, but more mundane things like strings, pointers to structures, etc.

For more information, see the following:

- **The OLE Memory Allocator**
- **Memory Management Rules**
- **Debugging Memory Allocations**

The OLE Memory Allocator

The COM Library provides an implementation of a memory allocator that is thread-safe (cannot cause problems in multi-threaded situations). Whenever ownership of an allocated chunk of memory is passed through a COM interface or between a client and the COM library, you must use this allocator to allocate the memory. Allocation internal to an object can use any allocation scheme desired, but the COM memory allocator is a handy, efficient, and thread-safe allocator.

A call to the API function **CoGetMalloc** provides a pointer to the OLE allocator, which is an implementation of the **IMalloc** interface. Rather than doing this, it is usually more efficient to call the helper functions **CoTaskMemAlloc**, **CoTaskMemAlloc**, and **CoTaskMemFree**, which wrap getting a pointer to the task memory allocator, calling the corresponding **IMalloc** method, and then releasing the pointer to the allocator.

Memory Management Rules

The life-time of pointers to interfaces is always managed through the **AddRef** and **Release** methods on every COM interface. For more information, refer to Rules for Managing Reference Counts.

For all other parameters, it is important to adhere to certain rules for managing memory. The following rules apply to all parameters of interface methods – including the return value – that are not passed by value:

- **in parameters** must be allocated and freed by the caller.
- **out parameter** must be allocated by the one called; freed by the caller using the standard COM task memory allocator. Refer to The OLE Memory Allocator for more information.
- **in-out parameter** is initially allocated by the caller, then freed and re-allocated by the one called, if necessary. As is true for out parameters, the caller is responsible for freeing the final returned value. The standard COM memory allocator must be used.

In the latter two cases, where one piece of code allocates the memory and a different piece of code frees it, using the COM allocator ensures that the two pieces of code are using the same allocation methods.

Another area that needs special attention is the treatment of out and in-out parameters in failure conditions. If a function returns a a failure code, the caller typically has no way to clean up the *out* or *in-out* parameters. This leads to a few additional rules:

Parameters must *always* be reliably set to a value that will be cleaned up without any action by the caller, in case of an error condition.

All out pointer parameters *must* explicitly be set to NULL. These are usually passed in a pointer-to-pointer parameter, but can also be passed as a member of a structure that the caller allocates and the called code fills. The most straightforward way to ensure this is (in part) to set these values to NULL on function entry. This rule is important, because it promotes more robust application interoperability.

Under error conditions, all in-out parameters must either be left alone by the code called (thus remaining at the value to which they were initialized by the caller) or be explicitly set, as in the out-parameter error return case.

Remember that these memory management conventions for COM applications apply only across public interfaces and APIs – there is no requirement at all that memory allocation strictly internal to a COM application need be done using these mechanisms.

Debugging Memory Allocations

OLE provides an interface for developers to use to debug their memory allocations: the **IMallocSpy** interface. For each method in **IMalloc**, there are two methods in **IMallocSpy**, a "pre" method and a "post" method. Once a developer implements it and publishes it to the system, the system calls the **IMallocSpy** "pre" method just before the corresponding **IMalloc** method, effectively allowing the debug code to "spy" on the allocation operation, and calling the "post" method to release the spy.

For example, when OLE detects that the next call is a call to **IMalloc::Alloc**, it would call **IMallocSpy::PreAlloc**, executing whatever debug operations the developer wants during the **Alloc** execution, and then, when the **Alloc** call returns, call **IMallocSpy::PostAlloc** to release the spy and return control to the code.

Processes and Threads

A process is a collection of virtual memory space, code, data, and system resources, while a thread is code that is to be serially executed within a process. A processor executes threads, not processes, so each 32-bit application has at least one process and one thread. Prior to the introduction of multiple threads of execution, applications were all designed to run on a single thread of execution. Processes communicate with one another through messages, using RPC to pass information between processes.

While COM defines three mulitiple-threading models, some information applies to threads and processes in general. A process always has at least one thread of execution, known as the *primary thread*, and can have multiple threads in addition to this. Once a thread begins to execute, it continues until it exits its routine, or until it is interrupted by a thread with higher priority, by a user action or by the kernel's thread scheduler. Each thread can run separate sections of code, or multiple threads can execute the same section of code. Threads executing the same block of code maintain separate stacks. Each thread in a process shares that process's global variables and resources.

The NT Scheduler determines when and how often to execute a thread according to a combination of the process's *priority class attribute* and the thread's *base priority*. You set a process's priority class attribute by calling the Win32 function **SetPriorityClass()**, and you set a thread's base priority with a call to **SetThreadPriority()**.

Multi-threaded applications must avoid two threading problems: deadlocks and races. A deadlock occurs when each thread is waiting for the other to do something. A race condition occurs when one thread finishes before another on which it depends, causing the former to use a bogus value because the latter has not yet supplied a valid one.

The OLE call control helps prevent deadlocks in calls between objects in different apartments. OLE supplies some functions specifically designed to help avoid race conditions in out-of-process servers; for information refer to **Out-of-process Server Implementation Helpers**.

While COM supports the single-thread-per-process model prevalent before the introduction of multiple threads of execution, writing code to take advantage of multiple threads make it possible to create more efficient applications than ever before by allowing a thread that is waiting for some time-consuming operation to allow another thread to be executed.

In general, the simplest way to view OLE's threading architecture is to think of all the COM objects in the process as divided into groups called *apartments*. A COM object lives in exactly one apartment, in the sense that its methods can legally be called directly only by a thread that belongs to that apartment. Any other thread that wants to call the object must go through a proxy.

There are two types of apartments: single-threaded apartments, and multi-threaded apartments.

Single-threaded Apartments – each thread that uses OLE is in a separate "apartment", and OLE synchronizes all incoming calls with the windows message queue. A process with a single thread of execution is simply a special case of this model.

Multi-threaded Apartments – Multiple threads in a single free-threaded apartment use OLE and calls to OLE objects are synchronized by the objects themselves.

A description of communication between single-threaded apartments and multi-threaded apartments within the same process is in **Single-/Multi-threaded Communication**.

Single-threaded apartments consist of exactly one thread, so all COM objects that live in a single-threaded apartment can receive method calls only from the one thread that belongs to that apartment. All method calls to a COM object in a single-threaded apartment are synchronized with the windows message queue for the single-threaded apartment's thread.

Multi-threaded apartments consist of one or more threads, so all COM objects that live in an multi-threaded apartment can receive method calls directly from any of the threads that belong to the multi-threaded apartment. Threads in a multi-theaded apartment use a model called "free-threading". OLE does not provide any synchronization of method calls to COM objects in an MTA. In particular, this means that the COM object must provide it's own synchronization if needed.

A process can have zero or more single-threaded apartments, and zero or one multi-threaded apartment. One way of looking at this has been the following:

- A process that consists of just one single-threaded apartment has been refered to as a *single-threaded process*
- A process that has two or more single-threaded apartments and no multi-threaded apartments has been called an *apartment model process*
- A process that has a multi-threaded apartment and no single-threaded apartments has been referred to as a *free-threaded process*
- A process that has a multi-threaded apartment and one or more single-threaded apartments as a *mixed model process*.

In reality, however, all process are apartment-model, it is just that some apartments have a single thread and some apartments have multiple threads. The threading model really applies to an apartment, not to a process. It can also apply to a class of objects, but it doesn't really apply to a component, such as a DLL, but to the object classes within the DLL. Different classes in a DLL can have different threading models.

In a process, the main apartment is the first to be initialized. In a single-threaded process, this remains the only apartment. Call parameters are marshaled between apartments, and OLE handles the synchronization through messaging. If you designate multiple threads in a process to be free-threaded, all free threads reside in a single apartment, parameters are passed directly to any thread in the apartment, and you must handle all synchronization. In a process with both free-threading and apartment threading, all free threads reside in a single apartment, and each apartment thread has an apartment to itself. A process that does OLE work is a collection of apartments with, at most, one multi-threaded apartment but any number of single-threaded apartments.

The threading models in OLE provide the mechanism for clients and servers that use different threading architectures to work together. Calls among objects with different threading models in different processes are naturally supported. From the perspective of the calling object, all calls to objects outside a process behave identically, no matter how the object being called is threaded. Likewise, from the perspective of the object being called, arriving calls behave identically, regardless of the threading model of the caller.

Interaction between a client and an out-of-process object is straightforward even when they use different threading models because the client and object are in different processes and OLE is involved in remoting calls from the client to the object. OLE, interposed between the client and the server, can provide the code for the threading models to interoperate, with standard marshaling and RPC. For example, if a single-threaded object is called simultaneously by multiple free-threaded clients, the calls will be synchronized by OLE by placing corresponding window messages in the server's message queue. The object's apartment will receive one call each time it retrieves and dispatches messages.

Some care must be taken to ensure that in-process servers interact properly with their clients. These issues are described in **In-process Server Threading Issues**.

The most important issue in programming with a multithreaded model is to ensure that the code is *thread-safe*, so messages intended for a particular thread go only to that thread, and access to threads is protected.

Choosing the Threading Model

Choosing the threading model for an object depends on the object's function. An object that does extensive I/O might support free-threading to provide maximum response to clients by allowing interface calls during I/0 latency. On the other hand, an object that interacts with the user might support apartment threading to synchronize incoming OLE calls with its window operations.

It is easier to support apartment threading in single-threaded apartments because OLE provides synchronization. Supporting free-threading is more difficult because the object must implement synchronization and thread local storage, but response to clients may be better because synchronization can be implemented for smaller sections of code. In single-threaded apartments, OLE provides synchronization on a per-call basis.

Single-threaded Apartments

Using single-threaded apartments (apartment model) offers a message-based paradigm for dealing with multiple objects running concurrently. It allows you to write more efficient code by allowing a thread that is waiting for some time-consuming operation to allow another thread to be executed.

Each thread in a process that is intialized as apartment-model, and which retrieves and dispatches window messages, is a single-threaded apartment thread. Each of these threads live within its own apartment. Within an apartment, interface pointers can be passed without marshaling. Thus, all objects in one single-threaded apartment thread communicate directly. Interface pointers must be marshaled when passed between apartments.

A logical grouping of related objects that all execute on the same thread, and so must have synchronous execution could live on the same single-threaded apartment thread. An apartment-model object cannot, however, reside on more than one thread. Calls to objects in other processes must be made within the context of the owning process, so distributed COM switches threads for you automatically when you call on a proxy.

The inter-process and inter-thread models are similar. When it is necessary to pass an interface pointer to an object in another apartment (on another thread) within the same process, you use the same marshaling model that objects in different processes use to pass pointers across process boundaries. By getting a pointer to the standard marshaling object, you can marshal interface pointers across thread boundaries (between apartments) in the same way you do between processes.

Rules for single-threaded apartments are simple, but it is important to follow them carefully:

- Every object should live on only one thread (within a single-threaded apartment).
- Initialize the COM library for each thread.
- Marshal all pointers to objects when passing them between apartments.
- Each single-threaded apartment must have a message loop to handle calls from other processes and apartments within the same process. Single-threaded apartments without objects (client only) also need a message loop to dispatch broadcast sendmessages that some applications use.
- DLL-based or in-process objects do not call the COM initialization functions; instead, they register their threading model with the **ThreadingModel** named-value under the **InprocServer32** key in the registry. Apartment-aware objects must also write DLL entry points carefully. There are special considerations that apply to threading in-process servers. For more information, see **In-process Server Threading Issues**.

While multiple objects can live on a single thread, no apartment-model object can live on more than one thread.

Each thread of a client process or out-of-process server must call **CoInitialize** or **OleInitialize**, which set the threading model to apartment, or call **CoInitializeEx**, and specify COINIT_APARTMENTTHREADED for the *dwCoInit* parameter. The main apartment is the thread that calls **CoInitialize** first. For information on in-process servers, refer to **In-process Server Threading Issues**.

All calls to an object must be made on its thread (within its apartment). It is forbidden to call an object directly from another thread; using objects in this free-threaded manner could cause problems for applications. The implication of this rule is that all pointers to objects must be marshaled when passed between apartments. OLE provides two functions for this purpose.
CoMarshalInterThreadInterfaceInStream, marshals an interface into a stream object that is returned to the caller, and **CoGetInterfaceAndReleaseStream** unmarshals an interface pointer from a stream object and releases it. These functions wrap calls to **CoMarshalInterface** and **CoUnmarshalInterface** functions, which require the use of the MSHCTX_INPROC flag.

In general, the marshaling is accomplished automatically by COM. For example, when passing an interface pointer as a parameter in a method call on a proxy to an object in another apartment, or when calling **CoCreateInstance**, COM does the marshaling automatically. However, in some special cases, where the application writer is passing interface pointers between apartments without using the normal COM mechanisms, the application writer must be aware that he is passing a pointer between apartments, and must handle the marshaling himself.

If one apartment (Apartment 1) in a process has an interface pointer and another apartment (Apartment 2) requires its use, Apartment 1 must call **CoMarshalInterThreadInterfaceInStream** to marshal the interface. The stream that is created by this function is thread-safe and must be stored in a variable that is accessible by Apartment 2. Apartment 2 must pass this stream to **CoGetInterfaceAndReleaseStream** to unmarshal the interface, and will get back a pointer to a proxy through which it can access the interface. The main apartment must remain alive until the client has completed all OLE work (because some in-process objects are loaded in the main-apartment, as described in **In-process Server Threading Issues**. After one object has been passed between threads in this manner, it is very easy to pass interface pointers as parameters. That way distributed COM does the marshaling and thread switching for the application.

To handle calls from other processes and apartments within the same process, each single-threaded apartment must have a message loop. This means that the thread's work function must have a **GetMessage/DispatchMessage** loop. If other synchronization primitives are being used to communicate between threads, the Win32 function **MsgWaitForMultipleObjects** can be used to wait for both messages and thread synchronization events. The Win32 SDK documentation for this function has an example of this sort of combination loop.

OLE creates a hidden window in each single-threaded apartment. A call to an object is received as a window message to this hidden window. When the object's apartment retrieves and dispatches the message, the hidden window will receive it. The window procedure will then call the corresponding interface method of the object.

When multiple clients call an object, the calls are queued in the message queue and the object will receive a call each time its apartment retrieves and dispatches messages. Because the calls are synchronized by OLE and the calls are always delivered by the thread that belongs to the object's apartment, the object's interface implementations need not provide synchronization. Single-threaded apartments can implement **IMessageFilter** to permit them to cancel calls or receive windows messages when necessary.

The object can be re-entered if one of its interface method implementations retrieves and dispatches messages or makes an ORPC call to another thread, thereby causing another call to be delivered to the object (by the same apartment). OLE does not prevent re-entrancy on the same thread but it provides thread safety. This is identical to the way in which a window procedure can be re-entered if it retrieves and dispatches messages while processing a message.

Multi-threaded Apartments

In a multi-threaded apartment, all the threads in the process that have been initialized as free-threading reside in a single apartment. Therefore, there is no need to marshal between threads. The threads need not retrieve and dispatch messages because OLE does not use window messages in this model.

Calls to methods of objects in the multi-threaded apartment can be run on any thread in the apartment. There is no serialization of calls - many calls may occur to the same method or to the same object simultaneously.Objects created in the multi-threaded apartment must be able to handle calls on their methods from other threads at any time.

Multi-threaded object concurrency offers the highest performance and takes the best advantage of multi-processor hardware for cross-thread, cross-process, and cross-machine calling, since calls to objects are not serialized in any way. This means, however, that the code for objects must provide synchronization in their interface implementations, typically through the use of Win32 synchronization primitives, such as *event objects*, *critical sections*, *semaphores*, or *mutexes*. In addition, because the object doesn't control the lifetime of the threads that are accessing it, no thread-specific state may be stored in the object (in Thread-Local-Storage).

OLE provides call synchronization for single-threaded apartments only. Multi-threaded apartments (containing free-threaded threads) do not receive calls while making calls (on the same thread). Multi-threaded apartments cannot make input synchronized calls. Asynchronous calls are converted to synchronous calls in multi-threaded apartments. The message filter is not called for any thread in a multi-threaded apartment.

To initialize a thread as free-threaded, call **CoInitializeEx**, specifying COINIT_MULTITHREADED. For information on in-process server threading, see **In-process Server Threading Issues**.

Multiple clients can call an object that supports free-threading simultaneously from different threads: In free threaded out-of-process servers, OLE creates a pool of threads in the server process and a client call (or multiple client calls) can be delivered by any of these threads at any time. An out-of-process server must also implement synchronization in its class factory. Free threaded, in-process objects can receive direct calls from multiple threads of the client.

The client can do OLE work in multiple threads. All threads belong to the same multi-threaded apartment. Interface pointers are passed directly from thread to thread within a multi-threaded apartment so interface pointers are not marshaled between its threads. Message filters (implementations of **IMessageFilter**) are not used in multi-threaded apartments. The client thread will suspend when it makes an OLE call to out-of-apartment objects and will resume when the call returns. Calls between processes are still handled by RPC.

Threads initialized with the free-threading model must implement their own synchronization. Win32 offers several means to do this: *events*, *critical sections*, *mutexes*, and *semaphores*. Following are brief descriptions of each – more information is available in the Win32 SDK.

A Win32 *event object* provides a way of signaling one or more threads that an event has occurred, essentially acting as a traffic cop. Any thread within a process can create an event object. A handle to the event is returned by the event-creating function **CreateEvent**. Once an event object has been created, threads with a handle to the object can wait on it before continuing execution.

A *critical section* is a section of code that requires *exclusive* access to some set of shared data before it can be executed. It may be used only by the threads within a single process. A critical section is like a turnstyle through which only one thread at a time may pass.

To ensure that no more than one thread at a time accesses shared data, a process's primary thread allocates a global CRITICAL_SECTION data structure and initializes its members. A thread entering a critical section calls the Win32 function **EnterCriticalSection()** and modifies the data structure's

members.

A thread attempting to enter a critical section calls **EnterCriticalSection** which checks to see whether the CRITICAL_SECTION data structure has been modified. If so, another thread is currently in the critical section, so the subsequent thread is put to sleep. A thread leaving a critical section calls **LeaveCriticalSection()**, which resets the data structure. When a thread leaves a critical section, WindowsNT wakes up one of the sleeping threads, which thereupon enters the critical section.

A mutex (short for *mutual exclusion*) object is a kernel object that performs the same function as a critical section, except that the mutex is accessible to threads running in different processes. Owning a mutex object is like having the floor in a debate. A process creates a mutex object by calling the Win32 function **CreateMutex()**, which returns a handle. The first thread requesting a mutex object obtains ownership of it. When the thread has finished with the mutex, ownership passes to other threads on a first-come, first-served basis.

A *semaphore* is a kernel object used to maintain a reference count on some available resource. A thread creates a semaphore for a resource by calling the Win32 function **CreateSemaphore()** and passing a pointer to the resource, an initial resource count, and the maximum resource count. This function returns a handle.

A thread requesting a resource passes its semaphore handle in a call to **WaitForSingleObject()**. The semaphore object polls the resource to determine if it is available. If so, the semaphore decrements the resource count and wakes the waiting thread. If the count is zero, the thread remains asleep until another thread releases a resource, causing the semaphore to increment the count to one.

Single-/Multi-threaded Communication

A client or server that supports both single and multi-threaded apartments will have one multi-threaded apartment, containing all threads initialized as free-threaded, and one or more single-threaded apartments. Interface pointers must be marshaled between apartments but can be used without marshaling within an apartment. Calls to objects in a single-threaded apartment will be synchronized by OLE. Calls to objects in the multi-threaded apartment will not be synchronized by OLE.

All of the information on single-threaded apartments applies to the threads marked as apartment model, and all of the information on multi-threaded apartments applies to all of the threads marked as free - threaded. Apartment threading rules apply to inter-apartment communication, requiring that interface pointers be marshaled between apartments with calls to **CoMarshalInterThreadInterfaceInStream** and **CoGetInterfaceAndReleaseStream**, as described in the **Single-threaded Apartments** section. For information on free-threading, see the **Multi-threaded apartments** section. Some special considerations apply when dealing with in-process servers, as described in **In-process Server Threading Issues**.

In-process server Threading Issues

An in-process server does not call **CoInitialize**, **CoInitializeEx**, or **OleInitialize** to mark its threading model. For thread-aware DLL-based or in-process objects, you need to set the threading model in the registry. The default model when you do not specify a threading model is single-thread-per-process. To specify a model, you add the **ThreadingModel** named-value to the **InprocServer32** key in the registry.

DLLs that support instantiation of a class object must implement and export the functions **DllGetClassObject** and **DllCanUnloadNow**. When a client wants an instance of the class the DLL supports, a call to **CoGetClassObject** (either directly or through a call to **CoCreateInstance**) calls **DllGetClassObject** to get a pointer to its class object when the object is implemented in a DLL. **DllGetClassObject** should therefore be able to give away multiple class objects or a single thread-safe object (essentially just using **InterlockedIncrement**/**InterlockedDecrement** on their internal reference count).

As its name implies, **DllCanUnloadNow** is called to determine whether the DLL that implements it is in use, so the caller can safely unload it if it is not. Calls to **CoFreeUnusedLibraries** from any thread always route through the main apartment's thread to call **DllCanUnloadNow**.

Like other servers, in-process servers can be single-threaded, apartment-threaded, free-threaded, or both. Each of these servers can be used by any OLE client, regardless of the threading model used by that client. There are certain considerations peculiar to client/inprocess-server interoperation.

All combinations of threading model interoperability are allowed between clients and in-process objects. Interaction between a client and an in-process object that use different threading models is exactly like the interaction between clients and out-of-process servers. For an in-process server, when the threading model of the client and inprocess server differ, OLE must interpose itself between the client and the object.

When an in-process object that supports the single threading model is be called simultaneously by multiple threads of a client, OLE cannot allow the client threads to directly access the object's interface because the object was not designed for such access. Instead OLE must ensure that calls are synchronized and are made only by the client thread that created the object. Therefore, OLE creates the object in the client's main apartment and requires all the other client apartments to access the object using proxies.

When a free-threaded apartment (multi-threaded apartment) in a client creates an apartment-threaded in-process server, OLE spins up a single-threaded apartment model 'host' thread in the client. This host thread will create the object and the interface pointer will be marshaled back to the client's free-threaded apartment. Similarly, when a single-threaded apartment in an apartment-model client creates a free threading in-process server, OLE spins up a free-threading host thread (multi-threaded apartmentP on which the object will be created and marshaled back to the client single-threaded apartment.

In general, if you design a custom interface on an in-process server, you should also provide the marshaling code for it so OLE can marshal the interface between client apartments.

OLE protects access to objects provided by a single-threaded DLL by requiring access from the same client apartment in which they were created. In addition, all of the DLL's entry points (like **DllGetClassObject** and **DllCanUnloadNow**) and global data should always be accessed by the same apartment. OLE creates such objects in the main apartment of the client, giving the main apartment direct access to the object's pointers. Calls from the other apartments use inter-thread marshaling to go from the proxy to the stub in the main apartment (using inter-thread marshaling) and then to the object. This allows OLE to synchronize calls to the object. Inter-thread calls are slow, so it is recommended that these servers be rewritten to support multiple apartments.

Like a single-threaded in-process server, an object provided by an apartment-model DLL must be

accessed by the same client apartment from which it was created. Objects provided by this server, however, can be created in multiple apartments of the client, so the server must implement its entry points (like **DllGetClassObject** and **DllCanUnloadNow**) for multi-threaded use. For example, if two apartments of a client try to create two instances of the in-process object simultaneously, **DllGetClassObject** can be called simultaneously by both apartments. **DllCanUnloadNow** must be written so the DLL is protected from being unloaded while code is still executing in the DLL.

If the DLL provides only one instance of the class factory to create all the objects, the class factory implementation must also be designed for multi-threaded use because it will be accessed by multiple client apartments. If the DLL creates a new instance of the class factory each time **DllGetClassObject** is called, the class factory need not be thread-safe.

Objects created by the class factory need not be thread-safe. Once created by a thread, the object is always accessed through that thread and all calls to the object are synchronized by OLE. The apartment-model apartment of a client that creates this object will get a direct pointer to the object. Client apartments which are different from the apartment in which the object was created must access the object through proxies. These proxies are created when the client marshals the interface between its apartments.

When an in-process DLL's **ThreadingModel** named-value is set to **Both**, an object provided by this DLL can be created and used directly (without a proxy) in single- or multi-threaded client apartments. However, it can only be used directly within the apartment in which it was created. To give the object to any other apartment, the object must be marshaled. The DLL's object must implement its own synchronization and can be accessed by multiple client apartments at the same time.

To speed performance for free-threaded access to in-process DLL objects, OLE provides the **CoCreateFreeThreadedMarshaler** function. This function creates a free-threaded marshaling object that can be aggregated with an in-process server object. When a client apartment in the same process needs access to an object in another apartment, aggregating the free threaded marshaler provides the client with a direct pointer to the server object, rather than to a proxy, when the client marshals the object's interface to a different apartment. The client does not need to do any synchronization. This works only within the same process – standard marshaling is used for a reference to the object that is sent to another process.

An object provided by in-process DLL that supports only free threading is a free-threaded object. It implements its own synchronization and can be accessed by multiple client threads at the same time. This server does not marshal interfaces between threads so this server can be created and used directly (without a proxy) only by multi-threaded apartments in a client. Single-threaded apartments that create it will access it through a proxy.

COM Clients and Servers

A critical part of COM is how clients and servers interact. A COM server is any object that provides services to clients. These services are in the form of implementations of COM interfaces that can be called by any client who is able to get a pointer to one of the interfaces on the server object. There are two main types of servers, in-process and out-of-process. In-process servers are implemented in a dynamic linked library (DLL), and out-of-process servers are implemented in an EXE file. Out-of-process servers can reside either on the local machine or on a remote machine.

A COM client is whatever code or object gets a pointer to a COM server, and uses its services by calling the methods of its interfaces.

The COM programming model and constructs have now been extended so that COM clients and servers can work together across the network, not just within a given machine. This has been done so existing applications can interact with new applications and with each other across networks with proper administration, while new applications can be written to take advantage of networking features.

In addition, client applications do not need to be aware of how server objects are packaged, whether they are packaged as in-process objects (in dynamic-link libraries), or as local or remote objects (in executables). Distributed COM further allows objects to be packaged as NT Services, synchronizing OLE with the rich administrative and system-integration capabilities of NT.

Also introduced are new features that complement existing OLE features with the security required to build distributed component software. For more information, refer to Security in COM_com_Security_in_COM.

With the increasing importance of distributed systems, COM has been extended to allow this location transparency to extend across a network for applications written for single machines, while adding features that extend these capabilities and add the security necessary in a network.

COM specifies a mechanism by which the class code can be used by many different applications.

For information on how COM enables client/server interaction, see the following:

- **Getting a Pointer to an Object**
- **Creating an Object through a Class Object**
- **COM Server Responsibilities**
- **Persistent Object State**
- **Security in COM**
- **Providing Class Information**
- **Inter-object Communication**
- **Call Synchronization**

Getting a Pointer to an Object

Because OLE does not have a strict class model, there are several ways to instantiate or to get a pointer to an interface on an object. There are, in fact, four methods through which a client obtains its first interface pointer to a given object:

- Call a COM Library API function that creates an object of a pre-determined type – that is, the function will only return a pointer to one specific interface for a specific object class.
- Call a COM Library API function that can create an object based on a class identifier (CLSID) and that returns any type of interface pointer requested.
- Call a method of some interface that creates another object (or connects to an existing one) and returns an interface pointer on that separate object.
- Implement an object with an interface through which other objects pass their interface pointer to the client directly.

For information on getting pointers to other interfaces on an object once you have the first one, see **QueryInterface: Navigating in an Object**.

There are numerous OLE functions that return pointers to specific interface implementations, such as **CoGetMalloc**, which retrieves a pointer to the standard OLE memory allocator. Most of these are helper functions, which retrieve a pointer to an OLE implementation of an interface on an object, as does **CoGetMalloc**. Most of these functions are described in the specific area they are related to, such as storage or data transfer.

There are several functions that, given a CLSID, a client can call to create an object instance and get a pointer to it. All of these functions are based on the function **CoGetClassObject**, which creates a class object and supplies a pointer to an interface that allows you to create instances of that class. While there must be information that says which system the server resides on, there is no need for that information to be contained in the client. The client needs to know only the CLSID, and never the absolute path of the server code. For more information, see **Creating an Object through a Class Object**.

Among the many interface methods that return a pointer to a separate object are several that create and return a pointer to an enumerator object, which allows you to determine how many items of a given type an object maintains. OLE defines interfaces for enumerating a wide variety of items, such as strings, several structures important in various OLE technologies, monikers, and **IUnknown** interface pointers. The typical way to create an enumerator instance and get a pointer to its interface is to call a method from another interface. For example, the **IDataObject** interface defines two methods, **EnumDAdvise** and **EnumFormatEtc**, that return pointers to interfaces on two different enumeration objects. There are many other examples in OLE of methods that return pointers to objects, such as the OLE Compound Document interface **IOleObject::GetClientSite**, which, when called on the embedded or linked object, returns a pointer to the container object's implementation of **IOleClientSite**.

The fourth way to get a pointer to an object is used when two objects, such as an OLE Compound Document container and server, need bi-directional communication. Each implements an object containing an interface method to which other objects can pass interface pointers. In the case of containers and servers, each object then passes its pointer to the other object. The implementing object, which is also the client of the created object, can then call the method and get the pointer that was passed.

Creating an Object through a Class Object

With the increasing importance of computer networks, it has become necessary for clients and servers to interact easily and efficiently, whether they reside on the same machine or across a network. Crucial to this is the ability of a client to be able to launch a server, create an instance of the server's object, and have access to the methods of the interfaces on the object.

OLE now provides extensions to this basic COM process that make it virtually seamless across a network. As before, if a client is able to identify the server through its CLSID, calling a few simple functions permit OLE to do all the work of locating and launching the server, and activating the object. New subkeys have been added to the registry that allow remote servers to register their location, so the client does not require that information. For applications that want to take advantage of networking features, new capabilities have been added to the object creation functions that allow more flexibility and efficiency.

OLE Class Objects and CLSIDs

A COM server is implemented as a COM class. A COM class is an implementation of a group of interfaces in code executed whenever you interact with a given object. There is an important distinction between a C++ class and a COM class. In C++, a class is a type. A COM class is simply a definition of the object, and carries no type, although a C++ programmer might implement it using a C++ class. COM is designed to allow a class to be used by different applications, including applications written without knowledge of that particular class's existence. Therefore, class code for a given type of object exists either in a dynamic linked library (DLL) or in another application (EXE).

Each COM class is identified by a CLSID, a unique 128-bit GUID, which the server must register. OLE uses this CLSID, at the request of a client, to associate specific data with the DLL or EXE containing the code that implements the class, thus creating an instance of the object. For information on registering a server, see **Registering COM Servers**, and **GUID Creation and Optimizations**.

For clients and servers on the same machine, the model previously supported, the CLSID of the server is all the client ever needs. On each machine, COM maintains a database (it makes use of the system registry on Windows platforms) of all the CLSIDs for the servers installed on the system. This is a mapping between each CLSID and the location of the DLL or EXE that houses the code for that CLSID. COM consults this database whenever a client wants to create an instance of a COM class and use its services, so the client never needs to know the absolute location of the code on the machine.

For distributed systems, COM provides registry entries that allow a remote server to register itself for use by a client. While applications need know only a server's CLSID, because they can rely on the registry to locate the server, COM allows clients to override registry entries and to specify server locations, to take full advantage of the network (see **Locating a Remote Object**).

The basic way to create an instance of a class is through a COM *class object*. This is simply an intermediate object that supports functions common to creating new instances of a given class. Most class objects used to create objects from a CLSID support the **IClassFactory** interface, an interface that includes the important method **CreateInstance**. You implement an **IClassFactory** interface for each class of object that you offer to be instantiated. For information on implementing IClassFactory, refer to **Implementing IClassFactory**.

> **Note** Servers that support some other custom class factory interface and call **CoGetClassObject** to get a pointer to that interface, and use it to create instances, are not required to support **IClassFactory** specifically. However, calls to activation functions other than **CoGetClassObject** (such as **CoCreateInstanceEx**) require that the server support **IClassFactory**.

When a client wants to create an instance of the server's object, it uses the desired object's CLSID in a call to **CoGetClassObject**. (This call can either be direct or implicit, through one of the object creation helper functions.) This function locates the code associated with the CLSID, and creates a class object, and supplies a pointer to the interface requested (**CoGetClassObject** takes a *riid* param that specifies the client's desired interface pointer).

With this pointer, the caller can create an instance of the object, and retrieve a pointer to a requested interface on the object. This is usually an initialization interface, used to activate the object (put it in the running state), so the client can do whatever work with the object that it wants to. Using these basic functions, the client must also take care to release all object pointers. OLE provides several helper functions that reduce the work of creating object instances. These are described in **Instance Creation Helper Functions**.

Locating a Remote Object

With the advent of COM for distributed systems, COM uses the basic model for object creation described in OLE Class Objects and CLSIDs, and adds more than one way to locate an object that may reside on another system in a network, without overburdening the client application.

COM has added registry keys that permit a server to register the name of the machine on which it resides, or the machine where an existing storage is located. Thus, client applications, as before, need know only the CLSID of the server.

However, for cases where it is desired, COM has replaced a previously reserved parameter of **CoGetClassObject** with a **COSERVERINFO** structure, which allows a client to specify the location of a server. Another important value in this function is the CLSCTX enumeration, which specifies whether the expected object is to be run in-process, out-of-process local, or out-of-process remote. Taken together, these two values and the values in the registry determine how and where the object is to be run. Instance creation calls, when they specify a server location, can override a registry setting. The algorithm OLE uses for doing this is described in the reference for the **CLSCTX** enumeration.

The client and server machines must both be members of domains with a trust relationship for all types of remote activation.

Instance Creation Helper Functions

In previous releases of OLE, the primary mechanism used to create an object instance was the **CoCreateInstance** function. This function encapsulates the process of creating a class object, using that to create a new instance and releasing the class object. Another function of this kind is the more specific **OleCreate**, the OLE Compound Document helper that creates a class object and retrieves a pointer to a requested object.

To smooth the process of instance creation on distributed systems, COM has introduced three important new instance creation functions:

- **CoCreateInstanceEx**
- **CoGetInstanceFromFile**
- **CoGetInstanceFromIStorage**

CoCreateInstanceEx extends **CoCreateInstance** to make it possible to create a single uninitialized object associated with the given CLSID on a specified remote machine. In addition, rather than requesting a single interface and obtaining a single pointer to that interface, **CoCreateInstanceEx** makes it possible to query for multiple interfaces and (if available) receive pointers to them in a single round trip, thus permitting fewer round trips between machines. This can make remote object interaction much more efficient. To do this, the function uses an array of **MULTI_QI** structures.

Creating an object through **CoCreateInstanceEx** still requires that the object be initialized through a call to one of the initialization interfaces (such as **IPersistStorage:::Load**). The two helper functions, **CoGetInstanceFromFile** and **CoGetInstanceFromIStorage** encapsulate both the instance creation power of **CoCreateInstanceEx** and initialization, the former from a file, and the latter from a storage.

COM Server Responsibilities

One of the most important ways for a client to get a pointer to an object is for the client to ask that a server be launched, and that an instance of the object provided by the server be created and activated. It is the responsibility of the server to ensure that this happens properly. There are several important parts to this.

The server must implement code for a class object through an implementation of either the **IClassFactory** or **IClassFactory2** interface.

The server must register its CLSID in the system registry on the machine on which it resides, and further, has the option of publishing its machine location to other systems on a network to allow clients to call it without requiring the client to know the server's location.

The server is primarily responsible for security – that is, for the most part, the server determines whether it will provide a pointer to one of its objects to a client.

In-process servers should implement and export certain functions that allow the client process to instantiate them.

This chapter contains the following sections:

Implementing IClassFactory

When a client uses a CLSID to request the creation of an object instance, the first step is creation of a class object, an intermediate object that contains an implementation of the methods of the **IClassFactory** interface. While OLE provides several instance creation functions, the first step in the implementation of these functions is the creation of a class object.

As a result, all servers must implement the methods of the **IClassFactory** interface. This interface contains two methods: **CreateInstance** and **LockServer**. **CreateInstance** must create an uninitialized instance of the object, and return a pointer to a requested interface on the object.

The **LockServer** method just increments the reference count on the class object to ensure that the server stays in memory, and does not shut down before the client is ready for it to do so.

To enable a server to be responsible for its own licensing, OLE defines **IClassFactory2**, which inherits its definition from **IClassFactory**. Thus, a server implementing **IClassFactory2** must, by definition, implement the methods of **IClassFactory**. For more information on **IClassFactory2**, see **Licensing and IClassFactory2**.

OLE also provides helper functions for implementing out-of-process servers. For more information, see **Out-of-process Server Implementation Helpers**.

Licensing and IClassFactory2

The **IClassFactory** interface on a class object provides the basic object creation mechanism of COM. Using **IClassFactory**, a server can control object creation on a machine basis. The implementation of the **IClassFactory::CreateInstance** method can allow or disallow object creation based the existence of a machine license. A machine license is a piece of information separate from the application that exists on a machine to indicate that the software was installed from a valid source, such as the vendor's installation disks. If the machine license does not exist, the server can disallow object creation. Machine licensing prevents piracy in cases where a user attempts to copy the software from one machine to another; because the license information is not copied with the software, and the machine that receives the copy is not licensed.

However, in a component software industry, vendors need a finer level of control over licensing. In addition to machine license control, the a vendor needs to allow some clients to create a component object while preventing other clients from the same capability. This kind of licensing requires that the client application obtain a license key from component while the client application is still under development. The client application uses the license key later at run-time to create objects on an unlicensed machine.

For example, if a vendor provides a library of controls to developers, the developer who purchases the library will have a full machine license, allowing the objects to be created on the development machine. The developer can then build a client application on the licensed machine incorporating one or more of the controls. When the resulting client application is run on another machine, the controls used in the client application must be created on the other machine even if that machine does not possess a machine license to the controls from the original vendor.

The **IClassFactory2** interface provides this level of control. To allow key-based licensing for any given component, you implement **IClassFactory2** on the class factory object for that component. **IClassFactory2** is derived from **IClassFactory**, so by implementing **IClassFactory2** the class factory object fulfills the basic COM requirements. **IClassFactory2** is defined as follows:

```
interface IClassFactory2 : IClassFactory
    {
    HRESULT GetLicInfo(LPLICINFO pLicInfo);
    HRESULT RequestLicKey(DWORD dwResrved, BSTR FAR* pbstrKey);
    HRESULT CreateInstanceLic(IUnknown *pUnkOuter
        , IUnknown *pUnkReserved, REFIID riid, BSTR bstrKey
        , void **ppvObject);
    };
```

The **GetLicInfo** method fills a **LICINFO** structure with information describing the licensing behavior of the class factory. For example, the class factory can provide license keys for run-time licensing if the *fRunTimeKeyAvail* member is TRUE.

The **RequestLicKey** method provides a license key for the component. A machine license must be available when the client calls this method.

The **CreateInstanceLic** method creates an instance of the licensed component if the license key parameter (**BSTR** *bstrKey*) is valid.

In its type information, a component uses the attribute **licensed** to mark the **coclass** that supports licensing through **IClassFactory2**.

To incorporate a licensed component into your client application, you use the methods in **IClassFactory2**.

First, you need a separate development tool that is also a client of the licensed component. The purpose

of this tool is to obtain the run-time license key and save it in your client application. This tool runs only on a machine that possesses a machine license for the component. The tool calls the **GetLicInfo** and **RequestLicKey** methods to obtain the run-time license key and then saves the license key in your client application. For example, the development tool could create a .H file containing the **BSTR** license key. Then, you would include that .H file in your client application.

To instantiate the component within your client application, you first try to instantiate the object directly with **IClassFactory::CreateInstance**. If **CreateInstance** succeeds, then the second machine is itself licensed for the component and objects can be created at will. If **CreateInstance** fails with the return code CLASS_E_NOTLICENSED, the only way to create the object is to pass the run-time key to the **CreateInstanceLic** method. **CreateInstanceLic** verifies the key and creates the object if the key is valid.

In this way an application built with components (such as controls), can run on a machine that has no other license—only the client application containing the run-time license is allowed to create the component objects in question.

The **IClassFactory2** interface supports flexibility in licensing schemes. For example, the server implementor can encrypt license keys in the component for added security. Server implementers can also enable or disable levels of functionality in their objects by providing different license keys for different functions. For example, one key might allow a base level of functionality, while another would allow basic and advanced functionality, and so on. See *OLE Controls Inside Out* published by MS Press for detailed consideration of these issues.

Registering COM Servers

After you have defined a class in code (ensuring that it implements **IClassFactory** or **IClassFactory2**) and assigned it a CLSID, you need to put information in the registry that will allow OLE, on request of a client with the CLSID, to create instances of its objects. This information tells the system, for a given CLSID, where the DLL or EXE code for that class is located, and how it is to be launched. There is more than one way of registering a class in the registry. In addition, there are other ways of "registering" a class with the system when it is running, so the system is aware that a running object is currently in the system. These topics are described in the following sections:

- **Registering a Class at Installation**
- **Registering a Running EXE Server**
- **Registering Objects in the ROT**
- **Self-registration**
- **Installing as a Win32 Service or User Account**

Registering a Class at Installation

If a class is intended to be available to clients at any time, as most applications are, you usually register it through an installation and setup program. This means putting information about the application into the registry, including how and where its objects are to be instantiated. This information must be registered for all CLSIDs. Other information is optional. Win32 tools, such as **Regsvr32**, make it simple to write a setup program that registers servers at installation.

If you are not relying on system defaults, there are two important keys in the registry: **CLSID** and **AppID**. Among the important pieces of information under these keys is how the object is to be instantiated. Objects can be designated as in-process, out-of-process local, or out-of-process remote.

Under the new **AppID** key, are several named-values that define information specific to that application. Among these are **RemoteServerName**, and **ActivateAtStorage**, both of which can be used to permit a client with no built-in knowledge of the location of the server, to create an object. For more information on remote instantiation, see Locating a Remote Object and Instance Creation Helper Functions.

A server can also be installed as a Win32 service, or to run under a specific user account. For more information, see Installing as a Win32 Service or User Account.

A server or ROT object that is not a Win32 service or run under a specific user account can be referred to as an "activate as activator" server. For these servers, the security context and the window station/desktop of the client must match the server's. A client attempting to connect to a remote server is considered to have a NULL window station/desktop, so only the server security context (for information on Windows NT SID, see the security section of the Win32 SDK) is compared in this instance. COM caches the window station/desktop of a process when the process first connects to the distributed COM service. Thus, COM clients and servers should not change their window station or thread desktops of the process after calling **CoInitialize** or **CoInitializeEX**.

When a class is registered as in-process, a call to **CoGetClassObject** to create its class object is automatically passed by OLE to the **DllGetClassObject** function, which the class must implement to give the calling object a pointer to its class object.

Classes implemented in executables can specify that COM should execute their process and wait for the process to register their class object's **IClassFactory** through a call to the **CoRegisterClassObject** function.

For detailed OLE registry information, see Registering Object Applications.

Registering a Running EXE Server

When an executable (EXE) server is launched, it should call **CoRegisterClassObject**, which registers the CLSID for the server in what is called the class table (this is a different table than the running object table). When a server is registered in the class table, it allows the SCM to determine that it is not necessary to launch the class again; because the server is already running. Only if the server is not listed in the class table will the SCM check the registry for appropriate values and launch the server associated with the given CLSID.

You pass **CoRegisterClassObject** the CLSID for the class and a pointer to its **IUnknown** interface. Clients who subsequently call **CoGetClassObject** with this CLSID will retrieve a pointer to their requested interface, as long as security does not forbid it. There are several instance creation and activation functions described in **Instance Creation Helper Functions**.

The server for a class object should call **CoRevokeClassObject** to revoke the class object (remove its registration) when all of the following are true:

- There are no existing instances of the object definition
- There are no locks on the class object
- The application providing services to the class object is not under user control (not visible to the user on the display).

Registering Objects in the ROT

Typically, when a client asks a server to create an object instance, the server typically creates moniker for the object, and registers it in the running object table (ROT) through a call to **IRunningObjectTable::Register**.

A few additional issues arise when registering ROT objects for use by remote clients. When the server calls **CreateFileMoniker** to create a file moniker to be registered in the ROT, servers should pass local file names that are drive-based, not in UNC format. This ensures that the moniker comparison data that is generated by the ROT register call will match what is used while doing a ROT lookup on the part of a remote client. This is because when the distribed COM service receives an activation request for a file local to the server from a remote client, the file is converted to a local-drive-based path.

Self-Registration

As component software continues to grow as a market, there will be more and more instances where a user obtains a new software component as a single DLL or EXE module, such as downloading a new component from an on-line service or receiving one from a friend on a floppy disk. In these cases, it is not practical to require the user to go through a lengthy installation procedure or setup program. Besides the licensing issues, which are handled through **IClassFactory2**, an installation procedure typically creates the necessary registry entries for a component to run properly in the COM and OLE context.

Self-Registration is the standard means through which a server module can package its own registry operations, both registration and unregistration, into the module itself. When used with licensing handled through **IClassFactory2**, a server can become an entirely self-contained module with no need for external installation programs or .REG files.

Any self-registering module, DLL or EXE, should first include a string called **OleSelfRegister** in the **StringFileInfo** section of its version information resource:

```
VS_VERSION_INFO VERSIONINFO

  ...

  BEGIN
    BLOCK "StringFileInfo"
     BEGIN
     #ifdef UNICODE
      BLOCK "040904B0" // Lang=US English, CharSet=Unicode
     #else
      BLOCK "040904E4" // Lang=US English, CharSet=Windows Multilingual
     #endif
       BEGIN
        ...
        VALUE "OLESelfRegister", "\0"
       END

    ...

    END

  ...

  END
```

The existence of this data allows any interested party, such as an application that wishes to integrate this new component, to determine if the server supports self-registration without having to attempt the self-registration process itself.

If the server is packaged in a DLL module, the DLL must export the functions **DllRegisterServer** and **DllUnregisterServer**. Any application that wishes to instruct the server to register itself (that is, all its CLSIDs and type library IDs) can obtain a pointer to **DllRegisterServer** through the Win32 API function **GetProcAddress**. Within **DllRegisterServer**, the DLL creates all its necessary registry entries, storing the correct path to the DLL for all **InprocServer32** or **InprocHandler32** entries.

When an application wishes to remove the component from the system, it should unregister that component by calling **DllUnregisterServer**. Within this call, the server removes exactly those entries it previously created in **DllRegisterServer**. The server should not blindly remove all entries for its classes because other software may have stored additional entries, such as a **TreatAs** key.

If the server is packaged in an EXE module, then the application wishing to register the server launches the EXE server with the command-line argument **/RegServer** or **-RegServer** (case-insensitive). If the application wishes to unregister the server, it launches the EXE with the command-line argument **/UnregServer** or **-UnregServer**. The self-registering EXE detects these command-line arguments and invokes the same operations as a DLL would within **DllRegisterServer** and **DllUnregisterServer**, respectively, registering its module path under **LocalServer32** instead of **InprocServer32** or **InprocHandler32**.

The server must register the full path to the installation location of the DLL or EXE module for their respective **InprocServer32**, **InprocHandler32**, and **LocalServer32** keys in the registry. The module path is easily obtained through the Win32 API function **GetModuleFileName**.

Installing as a Win32 Service or User Account

In addition to running as a *local server* executable (EXE), an OLE object may also package itself to run as a Win32 service when activated by a local or remote client. Win32 services support numerous useful and UI-integrated administrative features, including local and remote starting, stopping, pausing, and restarting, as well as the ability to establish the server to run under a specific user account and Window Station, and optionally to be interactive with the desktop.

An object written as a Win32 service is installed for use by OLE by establishing a **LocalService** named-value under its **AppID** key and performing a standard service installation (see the Win32 documentation and the SECSVR distributed COM sample application for more information about installation).

Classes may also be configured to run under a specific user account when activated by a remote client without being written as a Win32 service. To do this, the class installs a *user-name* and a *password* to be used when the SCM launches its *local server* process.

When a class is configured in this fashion, calls to **CoRegisterClassObject** with this CLSID will fail unless the process was launched by OLE on behalf of an actual activation request. In other words, classes configured to RunAs a particular user may not be registered under any other identity.

The user-name is taken from the **RunAs** named-value under the class's APPID key. If the user-name is "Interactive User", the class code is run in the security context of the currently logged on user and is connected to the interactive window station.

Otherwise, the password is retrieved from a secret and safe portion of the registry available only to administrators of the machine and to the system. The user-name and password are then used to create a logon-session in which the class code is run. When launched in this way, the class code runs with its own *desktop* and *window-station*, and does not share window-handles, the clipboard, or other UI elements with the interactive user or other classes running in other user accounts.

A server registered either with **LocalService** or **RunAs** can register an object in the running object table to allow any client to connect to it. To do so, the server's call to **IRunningObjectTable::Register** must set the ROTFLAGS_ALLOWANYCLIENT flag. A server setting this bit must have its executable name in the **AppID** section of the registry that refers to the AppID for the executable. An "activate as activator" server (not registered either as **LocalService** or **RunAs**) may not register an object with this flag.

Out-of-process Server Implementation Helpers

Four helper functions that can be called by out-of-process servers are now available to simplify the job of writing server code. OLE clients and OLE in-process servers typically would not call them. These functions are designed to help prevent race conditions in server activation when the servers have multiple apartments or multiple class objects. They can also, however, as easily be used for single-threaded and single class object servers. The functions are as follows:

- **CoAddRefServerProcess**
- **CoReleaseServerProcess**
- **CoSuspendClassObjects**
- **CoResumeClassObjects**

To shut down properly, an OLE server must keep track of how many object instances it has instantiated and how many times its **IClassFactory::LockServer** method has been called. Only when both of these counts reach zero, can a server shut down. In single-threaded OLE servers, the decision to shut down was coordinated with incoming activation requests by the fact that the requests were serialized by the message queue. The server, upon receiving a **Release** on it's final object instance and deciding to shut down, would revoke its class objects before any more activation requests were dispatched. If an activation request did come in after this point, OLE would recognize that the class objects were revoked, and would return an error to the SCM, which would then cause a new instance of the local server process to be run.

However, in an apartment model server, in which different class objects are registered on different apartments, and in all free-threaded servers, this decision to shut down must be co-ordinated with activation requests across multiple threads, so one thread of the server does not decide to shut down while another thread of the server is busy handing out class objects or object instances. One classical but cumbersome approach to solving this is to have the server, after it has revoked its class objects, recheck its instance count and stay alive until all instances have been released.

To make it easier for server writers to handle these types of race conditions, OLE provides two new reference counting functions. **CoAddRefServerProcess** increments a global per-process reference count. **CoReleaseServerProcess** decrements the global per-process reference count. When the global per-process reference count reaches zero, OLE automatically does a **CoSuspendClassObjects**, which prevents any new activation requests from comming in. The server can then deregister its various class objects from its various threads at leisure without worry that another activation request may come in. All new activation requests are henceforth handled by the SCM launching a new instance of the local server process.

The simplest way for a local server application to make use of these APIs is to call **CoAddRefServerProcess** in the constructor for each of its instance objects, and in each of its **IClassFactory::LockServer** methods when the *fLock* parameter is TRUE. The server application should also call **CoReleaseServerProcess** in the destructor of each of its instance objects, and in each of its *IClassFactory::LockServer* methods when the *fLock* parameter is FALSE.

Finally, the server application should pay attention to the return code from **CoReleaseServerProcess** and if it returns 0, the server application should initiate its cleanup, which, for a server with multiple threads, typically means that it should signal it's various threads to exit their message loops and call **CoRevokeClassObject** and **CoUninitialize**. Note that if these functions are used at all, they must be used in both the object instances and the **LockServer** method, otherwise, the server application may be shut down prematurely.

A slight change has been made in OLE for WindowsNT 4.0. When a **CoGetClassObject** request is made, OLE contacts the server, marshals the **IClassFactory** interface of the class object, returns to the client

process, unmarshals the **IClassFactory** interface, and returns this to the client. At this point, clients typically call **IClassFactory::LockServer**(TRUE) to prevent the server process from shutting down. However, there is a window of time between when the class object is marshaled and when the client calls **LockServer**, in which another client could connect to the same server, get an instance and **Release** that instance causing the server to shutdown and leaving the first client high and dry with a disconnected **IClassFactory** pointer. To prevent this race condition, OLE adds an implicit **IClassFactory::LockServer**(TRUE) to the class object when it marshals the **IClassFactory** interface, and an implicit **IClassFactory::LockServer**(FALSE) when the client releases the **IClassFactory** interface. Because of this change, it is no longer necessary to remote **LockServer** calls back to the server, so the proxy for **IClassFactory::LockServer** simply returns S_OK without actually remoting the call.

There is another activation-related race condition during initialization of an out-of-process server process. An OLE server that registers multiple classes typically calls **CoRegisterClassObject**(....REGCLS_LOCAL_SERVER) for each CLSID it supports. After it has done this for all classes, the server enters it's message loop. For a single-threaded OLE server, all activation requests are blocked until the server enters the message loop. However, for an apartment model server that registers different class objects in different apartments, and for all free-threaded servers, activation requests can arrive earlier than this. In the case of apartment model servers, activation requests could arrive as soon as any one thread has entered its message loop. In the case of free-threaded servers, an activation request could arrive as soon as the first class object is registered. Since an activation can happen this early, it is also possible for the final **Release** to occur (and hence cause the server to begin shutting down) before the rest of the server has had a chance to finish initializing.

To eliminate these race conditions and simplify the job of the server writer, any server that wants to register multiple class objects with OLE should call **CoRegisterClassObject**(...., REGCLS_LOCAL_SERVER | REGCLS_SUSPENDED) for each different CLSID the server supports. After all classes have been registered and the server process is ready to accept incoming activation requests, the server should make one call to **CoResumeClassObjects**. This API tells OLE to inform the SCM about all the registered classes, and it begins letting activation requests into the server process. Using these APIs has serveral advantages. First, only one call is made to the SCM regardless of how many CLSIDs are registered, thus reducing the overall registration time (and hence startup time of the server application). The second advantage is that if the server has multiple apartments and different CLSIDs are registered in different apartments, or if the server is a free-threaded server, no activation requests will come in until the server calls **CoResumeClassObjects**, giving the server a chance to register all of its CLSIDs and get properly set up before having to deal with activation requests, and possible shut down requests.

GUID Creation and Optimizations

Because a CLSID, like an interface identifier (IID), is a GUID, no other class, no matter who writes it, has a duplicate CLSID. Server implementers generally obtain CLSIDs through the **CoCreateGUID** function in COM. This function is guaranteed to produce unique CLSIDs, so server implementors across the world can independently develop and deploy their software without fear of accidental collision with software written by others.

Using unique CLSIDs avoids the possibility of name collisions among classes because CLSIDs are in no way connected to the names used in the underlying implementation. So, for example, two different vendors can write classes called "StackClass," but each would have a unique CLSID and therefore could not be confused.

Both COM and OLE frequently must map GUIDs (IIDs and CLSIDs) to some arbitrarily large set of other values. As an application developer, you can help speed up such searches, and thereby enhance system performance, by generating the GUIDs for your application as a block of consecutive values.

The most efficient way to generate a block of consecutive GUIDs is to run the uuidgen utility using the /n switch, which generates a block of UUIDs, each of whose first DWORD value is incremented by one. (For more information on using the uuidgen utility, see "The uuidgen Utility.")

For example, if you were to type

```
uuidgen -n5 -s >guids.txt
```

the uuidgen utility would generate a block of UUIDs similar to the following:

```
{12340001-4980-1920-6788-123456789012}
{12340002-4980-1920-6788-123456789012}
{12340003-4980-1920-6788-123456789012}
{12340004-4980-1920-6788-123456789012}
{12340005-4980-1920-6788-123456789012}
```

One method for generating and tracking GUIDs for an entire project begins with generating a block of some arbitrarily large number of UUIDs – say, 500. For example, if you were to type

```
uuidgen -n500 -s >guids.txt
```

the utility would generate 500 consecutive UUIDs and write them to the specified text file. You could then check this file into your source tree, providing a single repository for all GUIDs to be used in a project. As people require GUIDs for their portions of the project, they can check out the file, take however many GUIDs they need, marking them as taken and leaving a note about where in the code or "spec" they are using them.

In addition to improving system performance, generating blocks of consecutive GUIDs in this way has the following benefits:

- A central file containing all GUIDs for an application makes it easy to keep track of which GUIDs are for what and which people are using them.
- A block of consecutive GUIDs associated with a particular application helps developers and testers recognize internal GUIDs during debugging and makes it easier to find them in the system registry because they are stored sequentially.

Persistent Object State

Some COM objects can save their internal state when asked to do so by a client. COM defines standards through which clients can request objects to be initialized, loaded, and saved to and from a data store (for example, flat file, structured storage, or memory). It is the client's responsibility to manage the *place* where the object's persistent data is stored, but not the *format* of the data. COM objects that adhere to these standards are called *persistent objects.* See **Persistent Object Interfaces** and **Initializing Persistent Objects** for more information.

Persistent Object Interfaces

A persistent object implements one or more *persistent object interfaces.* Clients use persistent object interfaces to tell those objects when and where to store their state. All persistent object interfaces are derived from **IPersist**, so any object that implements any persistent object interface also implements **IPersist**.

The following persistent object interfaces are currently defined:

IPersistStream

IPersistStreamInit

IPersistStorage

IPersistFile

IPersistMoniker

IPersistMemory

IPersistPropertyBag

Implementers choose which persistent object interfaces an object supports depending on how the object is to be used. By not supporting any persistent object interfaces, the implementer is effectively saying, "This object's state cannot be persistently stored." By supporting one or more persistent object interfaces, the implementer is effectively saying, "This object's state can be persistently stored in one or more data store mediums."

For example, several object types allow support for different persistent object interfaces:

Category	Persistent Object Interfaces Typically Supported
Monikers	**IPersistStream**
OLE embeddable objects	**IPersistStorage**, **IPersistFile**
ActiveX™ controls	**IPersistStreamInit**, **IPersistStorage**, **IPersistMemory**, **IPersistPropertyBag**, **IPersistMoniker**
ActiveX document objects	**IPersistStorage**, **IPersistFile**

Client implementers can also choose which persistent object interfaces the client can use. The interfaces a client uses is usually determined by where the client can store its own data. A client that can store its data only in a flat file will probably use only **IPersistStreamInit**, **IPersistMoniker**, and **IPersistPropertyBag**. (**IPersistStreamInit** can replace **IPersistStream** in most applications, because it contains that definition and adds an initialization method). A client that can save its data to a structured storage file will, in addition, use **IPersistStorage**.

Initializing Persistent Objects

Several of the persistent object interfaces (**IPersistStreamInit**, **IPersistStorage**, **IPersistMemory**, and **IPersistPropertyBag**) allow clients to initialize objects to a "fresh" or "default" state. This initial state is different from that of a newly created object, which has no state. Initializing an object's state, even to the default state, may be a compute- or resource-intensive operation. By separating creation from initialization, the initialization can be performed only when it is actually needed, and clients can avoid initializing objects to the default state only to immediately load previously stored data.

Security in COM

Prior to Windows NT 4.0, there was no special support for security in OLE beyond that provided by the operating system, so out-of-process servers had the same permissions as the interactive user, and an object could be instantiated for any CLSID in the registry, no matter who the user was.

To make it possible to implement an object that could perform privileged operations without compromising security, security features, which use and enhance the Windows NT security model, have been added to OLE. There are two main areas:

- **Launch Security**
- **Call Security**

Launch Security controls which objects a client is allowed to instantiate. *Call Security* dictates how security operates at the call level between an established connection from a client to a server. While anyone can get interface pointers from the class table, they cannot use them if they do not have call permissions.

OLE provides a default security model, but also defines call-level interfaces that external security providers can implement to control object security.

It is also possible to have a server run as a given user account, through setting the **RunAs** named-value. This can be used to restrict or enhance available operations. For more information, see Installing as a Win32 Service or User Account.

The remainder of this section describes the capabilities of COM security in greater detail.

Launch Security

Activation security controls which classes a client is allowed to launch and retrieve objects from. Launch security is automatically applied by the Service Control Manager (SCM) of a particular machine. Upon receipt of a request from a remote client to activate an object (as described in Instance Creation Helper Functions), the SCM of the machine checks the request against activation security information stored within its registry.

There are two machine-wide secure settings in the registry, to which only machine administrators and the system have full access. All other users have only read-access. These are **EnableDCOM** and **DefaultLaunchPermission**. The **EnableDCOM** allows or disallows remote clients to launch class code and connect to objects for the system, and **DefaultLaunchPermission**, as the name implies, sets the default Access Control List (ACL) of who has permission to classes on the system.

You can override the default for any given class by assigning the desired permissions to the **LaunchPermission** key.

Call Security

COM provides two mechanisms to secure calls. The first is done automatically by the COM infrastructure. If the application provides some setup information, COM will make all the necessary checks to secure the application's objects. This automatic mechanism does security checking for the process, not for individual objects or methods. The second is a set of functions and interfaces that applications may use to do their own security checking, and provide more fine-grained security. Furthermore, the two mechanisms are not exclusive: an application may ask COM to perform automatic security checking and also perform its own.

COM call security services are divided into three categories:

- General functions called by both clients and servers
- New interfaces on client proxies and related helper functions
- Server-side functions and call-context interfaces.

If you are using the default security values for a process for authentication and authorization, no security initialization call is necessary. If, however, you want to set other values for that process, you would call **CoInitializeSecurity**. This both initializes and registers these values. The values set in this call then become the default values for that process. The proxy interfaces allow the client to control the security on calls to individual interfaces.

The **IClientSecurity** interface is implemented locally to the client by the interface remoting layer. The client calls its methods to control the security of individual interface proxies on the object prior to making a call on one of the interfaces. Generally, clients using the default implementation instead call the helper functions that access that implementation and simplify the code: **CoQueryProxyBlanket**, **CoSetProxyBlanket**, and **CoCopyProxy**. Calling **IClientSecurity** directly is slightly more efficient then calling the helper functions. **IClientSecurity** works with all authentication services (NTLMSSP, DEC, kerberos). Some custom marshalled objects might not support **IClientSecurity**.

The server APIs and interfaces allow the server to retrieve security information about a call and to impersonate the caller. The **IServerSecurity** interface is implemented for all providers, but may be absent for some custom-marshaled interfaces. Helper functions are also available that rely on the **IServerSecurity** interface implementation: **CoQueryClientBlanket**, **CoImpersonateClient**, and **CoRevertToSelf**.

In a typical scenario, the client queries an existing object for **IClientSecurity**, which is implemented locally by the interface remoting layer. The client uses **IClientSecurity** to control the security of individual interface proxies on the object prior to making a call on one of the interfaces. When a call arrives at the server, the server may call **CoGetCallContext** to retrieve an **IServerSecurity** interface. **IServerSecurity** allows the server to check the client's authentication and to impersonate the client, if needed. The **IServerSecurity** object is valid for the duration of the call. **CoInitializeSecurity** allows the client to establish default call security for the process, avoiding the use of **IClientSecurity** on individual proxies. **CoInitializeSecurity** allows a server to register automatic authentication services for the process.

Implementations of **QueryInterace** must never check ACLs. COM requires that an object which supports a particular IID always return success when queried for that IID. Aside from the requirement, checking ACLs on **QueryInterface** does not provide any real security. If client A legally has access to interface IFoo, A can hand it directly to B without any calls back to the server. Additionally, OLE caches interface pointers and will not call **QueryInterface** on the server every time a client does a query. For more information on implementing **QueryInterface**, see **Rules for Implementing QueryInterface**.

Machine administrators and the system only have full access to the portion of the registry that contains the default machine-wide call security settings. All other uses have read access only. These named values are used for classes that do not call **CoInitializeSecurity**, and are as follows:

- **DefaultAccessPermission**
- **LegacyAuthenticationLevel**
- **LegacyImpersonationLevel**
- **LegacyMutualAuthentication**
- **LegacySecureReferences**

To set access to objects of a specific class, there is the single named-value, **AccessPermission**

Providing Class Information

It is often useful for a client of an object to examine the object's type information. Given the object's CLSID, a client can locate the object's type library using registry entries, and then can scan the type library for the **coclass** entry in the library matching the CLSID.

However, not all objects have a CLSID, although they still need to provide type information. In addition, it is convenient for a client to have a way to simply ask an object for its type information instead of going through all the tedium to extract the same information from registry entries.

This capability is important when dealing with outgoing interfaces on connectable objects. See Using IProvideClassInfo in the Connectable Objects chapter for more information on how connectable objects provide this capability.

In these cases, a client can query the object for any of the **IProvideClassInfo[*x*]** interfaces. If these interfaces exist, the client calls **IProvideClassInfo[*x*]::GetClassInfo** to get the type information for the interface.

By implementing **IProvideClassInfo[*x*]**, an object specifies that it can provide type information for its entire class, that is, what it would describe in its **coclass** section of its type library, if it has one. The **GetClassInfo** method returns an **ITypeInfo** pointer corresponding to the object's **coclass** information. Through this **ITypeInfo** pointer, the client can examine all the object's incoming and outgoing interface definitions.

The object can also provide **IProvideClassInfo2**. The **IProvideClassInfo2** interface is a simple extension to **IProvideClassInfo** that makes it quick and easy to retrieve an object's outgoing interface identifiers for its default event set. **IProvideClassInfo2** is derived from **IProvideClassInfo**.

Inter-object Communication

COM is designed to allow clients to communicate *transparently* with objects, regardless of where those objects are running – the same process, the same machine, or a different machine. This provides a *single programming model* for all types of objects, both object clients and object servers.

From a client's point of view, all objects are accessed through interface pointers. A pointer must be in-process. In fact, any call to an interface function always reaches *some* piece of in-process code first. If the object is in-process, the call reaches it directly, with no intervening system-infrastructure code. If the object is out-of-process, the call first reaches what is called a "proxy" object provided either by COM itself or by the object (if the implementor wishes). The proxy packages call parameters (including any interface pointers) and generates the appropriate remote procedure call (or other communication mechanism in the case of custom generated proxies) to the other process or the other machine where the object implementation is located. This process of packaging pointers for transmission across process boundaries is called *marshaling*.

From a server's point of view, all calls to an object's interface functions are made through a pointer to that interface. Again, a pointer only has context in a single process, and so the caller must always be some piece of in-process code. If the object is in-process, the caller is the client itself. Otherwise, the caller is a "stub" object provided either by COM or by the object itself. The stub receives the remote procedure call (or other communication mechanism in the case of custom generated proxies) from the "proxy" in the client process, *unmarshals* the parameters, and calls the appropriate interface on the server object. From the points of view of both clients and servers, they always communicate directly with some other in-process code.

OLE provides an implementation of marshaling, referred to as *standard marshaling*. This implementation works very well for most objects, and greatly reduces programming requirements, making the marshaling process effectively transparent.

The clear separation of interface from implementation of OLE's process transparency can, however, get in the way for some situations when performance is of critical concern. The design of an interface that focuses on its function from the client's point of view can sometimes lead to design decisions that conflict with efficient implementation of that interface across a network. In cases like this, what is needed is not pure process transparency, but "process transparency, unless you need to care." COM provides this capability by allowing an object implementor to support *custom marshaling.* Standard marshaling is, in fact, an instance of custom marshaling – it is the default implementation used when an object does not require custom marshaling.

You can implement custom marshaling to allow an object to take different actions when used from across a network than it takes under local access – and it is completely transparent to the client. This architecture makes it possible to design client/object interfaces without regard to network performance issues, then, later, to address network performance issues without disrupting the established design.

COM does not specify how components are structured; it specifies how they interact. COM leaves the concern about the internal structure of a component to programming languages and development environments. Conversely, programming environments have no set standards for working with objects outside of the immediate application. C++, for example, works extremely well to manipulate objects inside an application, but has no support for working with objects outside the application. Generally, all other programming languages are the same in this regard. Therefore COM, through language-independent interfaces, picks up where programming languages leave off to provide the network-wide interoperability.

The double indirection of the *vtbl* structure means that the pointers in the table of function pointers do not need to point directly to the real implementation in the real object. This is the heart of process transparency.

For in-process servers, where the object is loaded directly into the client process, the function pointers in

the table point directly to the actual implementation. In this case, a function call from the client to an interface method directly transfers execution control to the method. This cannot, however, work for local, let alone remote, objects because pointers to memory cannot be shared between processes. Nevertheless, the client must be able to call interface methods *as if it were calling the actual implementation*. Thus, the client uniformly transfers control to a method in some object by making the call.

A client always calls interface methods in some in-process object. If the actual object is local or remote, the call is made to a proxy object which then makes a remote procedure call to the actual object.

So what method is actually executed? The answer is that whenever there is a call to an out-of-process interface, each interface method is implemented by a proxy object. The proxy object is always an in-process object that acts on behalf of the object being called. This proxy object knows that the actual object is running in a local or remote server.

The proxy object packages up the function parameters in some data packets and generates an RPC call to the local or remote object. That packet is picked up by a stub object in the server's process on the local or a remote machine, which unpacks the parameters and makes the call to the real implementation of the method. When that function returns, the stub packages up any out-parameters and the return value, sends it back to the proxy, which unpacks them and returns them to the original client.

Thus, client and server always talk to each other as if everything was in-process. All calls from the client and all calls to the server are, at some point, in-process. But because the *vtbl* structure allows some agent, like COM, to intercept all function calls and all returns from functions, that agent can redirect those calls to an RPC call as necessary. Although in-process calls are, naturally, faster than out-of-process calls, the process differences are completely transparent to the client and server.

Marshaling Details

Marshaling is the process of packaging and unpackaging parameters so a remote procedure call can take place. Different parameter types are marshaled in different ways. For example, marshaling an integer parameter involves simply copying the value into the message buffer (although even in this simple case, there are issues such as byte ordering to deal with in cross-machine calls). Marshaling an array, however, is a more complex process. Array members are copied in a specific order so that the other side can reconstruct the array exactly. When a pointer is marshaled, the data that the pointer is pointing to is copied following rules and conventions for dealing with nested pointers in structures. Unique functions exist to handle the marshaling of each parameter type.

With standard marshaling, the proxies and stubs are system-wide resources for the interface, and they interact with the channel through a standard protocol. Standard marshaling can be used by both standard OLE interfaces and custom interfaces:

- In the case of most OLE interfaces, the proxies and stubs for standard marshaling are themselves in-process component objects which are loaded from a system-wide DLL provided by OLE in OLE32.DLL.
- In the case of custom interfaces, the proxies and stubs for standard marshaling are generated by the interface designer, typically with MIDL. These proxies and stubs are statically configured in the registry, so any potential client can use the custom interface across process boundaries. These proxies and stubs are loaded from a DLL that is located via the system registry using the interface ID (IID) for the custom interface they marshal.

As an alternative to standard marshaling, an interface (standard or custom) can use custom marshaling. With custom marshaling, an object dynamically implements the proxies at run-time for each interface that it supports. For any given interface, the object can select OLE-provided standard marshaling or custom marshaling. This choice is made by the object on an interface by interface basis. Once the choice is made for a given interface, it remains in effect during the object's lifetime. However one interface on an object can use custom marshaling while another uses standard marshaling.

Custom marshaling is inherently unique to the object that implements it. It uses proxies implemented by the object and provided to the system on request at run-time. Objects that custom-marshal must implement the **IMarshal** interface, whereas objects that support standard marshaling do not.

If you decide to write a custom interface, you must provide marshaling support for it. Typically, you will provide a standard marshaling DLL for the interface you design. You can use the tools contained in the *Win32 SDK* CD to create the proxy/stub code and the proxy/stub DLL.

For a client to make a call to an interface method in an object in another process involves the cooperation of several components. The standard proxy is a piece of interface-specific code that resides in the client's process space and prepares the interface parameters for transmittal. It packages, or marshals, them in such a way that they can be re-created and understood in the receiving process. The standard stub, also a piece of interface-specific code, resides in the server's process space and reverses the work of the proxy. The stub unpackages, or unmarshals, the sent parameters and forwards them to the object application. It also packages reply information to send back to the client.

Note Readers more familiar with RPC than OLE may be used to seeing the terms client stub and server stub. These terms are analogous to proxy and stub.

The following diagram shows the flow of communication between the components involved. On the client side of the process boundary, the client's method call goes through the proxy and then onto the channel. Note that the channel is part of the COM library. The channel sends the buffer containing the marshaled parameters to the RPC run-time library, which transmits it across the process boundary. The RPC run-

time and the COM libraries exist on both sides of the process. Note also that the distinction between the channel and the RPC run-time is a characteristic of this implementation and is not part of the programming model or the conceptual model for OLE client/server objects. OLE servers see only the proxy or stub and, indirectly, the channel. Future implementations may use different layers below the channel or no layers.

{ewc msdncd, EWGraphic, bsd23513 0 /a "SDK.WMF"}

Components of interprocess communications

Proxy

A proxy resides in the address space of the calling process and acts as a surrogate for the remote object. From the perspective of the calling object, the proxy is the object. Typically, the proxy's role is to package the interface parameters for calls to methods in its object interfaces. The proxy packages the parameters into a message buffer and passes the buffer onto the channel, which handles the transport between processes. The proxy is implemented as an aggregate, or composite, object. It contains a system-provided, manager piece called the proxy manager and one or more interface-specific components called interface proxies. The number of interface proxies equals the number of object interfaces that have been exposed to that particular client. To the client complying with the component object model, the proxy appears to be the real object.

> **Note** With custom marshaling, the proxy can be implemented similarly or it can communicate directly with the object without using a stub.

Each interface proxy is a component object that implements the marshaling code for one of the object's interfaces. The proxy represents the object for which it provides marshaling code. Each proxy also implements the **IRpcProxyBuffer** interface. Although the object interface represented by the proxy is public, the **IRpcProxyBuffer** implementation is private and is used internally within the proxy. The proxy manager keeps track of the interface proxies and also contains the public implementation of the controlling **IUnknown** interface for the aggregate. Each interface proxy can exist in a separate DLL that is loaded when the interface it supports is materialized to the client.

The following diagram shows the structure of a proxy that supports the standard marshaling of parameters belonging to two interfaces: **IFoo1** and **IFoo2**. Each interface proxy implements **IRpcProxyBuffer** that is used for internal communication between the aggregate pieces. When the proxy is ready to pass its marshaled parameters across the process boundary, it calls methods in the **IRpcChannelBuffer** interface, which is implemented by the channel. The channel in turn forwards the call to the RPC run-time library so that it can reach its destination in the object.

{ewc msdncd, EWGraphic, bsd23513 1 /a "SDK.WMF"}

Structure of the Proxy

Stub

The stub, like the proxy, is made up of one or more interface pieces and a manager. Each interface stub provides code to unmarshal the parameters and code that calls one of the object's supported interfaces. Each stub also provides an interface for internal communication. The stub manager keeps track of the available interface stubs.

There are, however, some differences between the stub and the proxy:

- The most important difference is that the stub represents the client in the object's address space.
- The stub is not implemented as an aggregate object since there is no requirement that the client be viewed as a single unit; each piece in the stub is a separate component.
- The interface stubs are private, rather than public.
- The interface stubs implement **IRpcStubBuffer**, not **IRpcProxyBuffer**.
- Instead of packaging parameters to be marshaled, the stub unpackages them after they have been marshaled and then packages the reply.

The following diagram shows the structure of the stub. Each interface stub is connected to an interface on the object. The channel dispatches incoming messages to the appropriate interface stub. All the components talk to the channel through **IRpcChannelBuffer**, the interface that provides access to the RPC run-time library.

{ewc msdncd, EWGraphic, bsd23513 2 /a "SDK.WMF"}

Structure of the Stub

Channel

The channel has the responsibility of transmitting all messages between client and object across the process boundary. The channel has been designed to work transparently with different channel types, is compatible with OSF DCE standard RPC, and supports single and multi-threaded applications.

Microsoft RPC

RPC is a model for programming in a distributed computing environment. The goal of RPC is to provide transparent communication so that the client appears to be directly communicating with the server. Microsoft's implementation of RPC is compatible with the Open Software Foundation (OSF) Distributed Computing Environment (DCE) RPC.

You can configure RPC to use one or more transports, one or more name services, and one or more security servers. The interface to those providers are handled by RPC. Because Microsoft RPC is designed to work with multiple providers, you can choose the providers that work best for your network. The transport is responsible for transmitting the data across the network. The name service takes an object name, such as a moniker, and finds its location on the network. The security server offers applications the option of denying access to specific users and/or groups. Refer to the section Interface Design for more detailed information about application security.

In addition to the RPC run-time libraries, Microsoft RPC includes the Interface Definition Language (IDL) and its compiler. Although the IDL file is a standard part of RPC, Microsoft has enhanced it to extend its functionality to support custom COM interfaces. The Microsoft Interface Definition Language (MIDL) compiler uses the IDL file that describes your custom interface to generate several files discussed in the section "Building a Proxy/Stub DLL."

Call Synchronization

OLE applications must be able to deal correctly with user input while processing one or more calls from OLE or the operating system. OLE provides call synchronization for single-threaded apartments only. Multi-threaded apartments (containing free-threaded threads) do not receive calls while making calls (on the same thread). Multi-threaded apartments cannot make input synchronized calls. Asynchronous calls are converted to synchronous calls in multi-threaded apartments. The message filter is not called for any thread in a multi-threaded apartment. For more information on threading issues, see **Processes and Threads**.

OLE calls between processes fall into three categories:

- Synchronous calls
- Asynchronous notifications
- Input-synchronized calls

Most of the communication that takes place within OLE is synchronous. When making *synchronous calls,* the caller waits for the reply before continuing and can receive incoming messages while waiting. OLE enters a modal loop to wait for the reply, receiving and dispatching other messages in a controlled manner.

When sending *asynchronous notifications,* the caller does not wait for the reply. OLE uses **PostMessage** or high-level events to send asynchronous notifications, depending on the platform. OLE defines five asynchronous methods:

- **IAdviseSink::OnDataChange**
- **IAdviseSink::OnViewChange**
- **IAdviseSink::OnRename**
- **IAdviseSink::OnSave**
- **IAdviseSink::OnClose**

While OLE is processing an asynchronous call, synchronous calls cannot be made. For example, a container application's implementation of **IAdviseSink::OnDataChange** cannot contain a call to **IPersistStorage::Save**.

When making *input-synchronized calls,* the object called must complete the call before yielding control. This ensures that focus management works correctly and that data entered by the user is processed appropriately. These calls are made by OLE through the Windows **SendMessage** function, without entering a modal loop. While processing an input-synchronized call, the object called must not call any function or method (including synchronous methods) that might yield control.

The following methods are input synchronized:

- **IOleWindow::GetWindow**
- **IOleInPlaceActiveObject::OnFrameWindowActivate**
- **IOleInPlaceActiveObject::OnDocWindowActivate**
- **IOleInPlaceActiveObject::ResizeBorder**
- **IOleInPlaceUIWindow::GetBorder**
- **IOleInPlaceUIWindow::RequestBorderSpace**
- **IOleInPlaceUIWindow::SetBorderSpace**
- **IOleInPlaceFrame::SetMenu**

- **IOleInPlaceFrame::SetStatusText**
- **IOleInPlaceObject::SetObjectRects**

To minimize problems that can arise from asynchronous message processing, the majority of OLE method calls are synchronous. With synchronous communication, there is no need for special code to dispatch and handle incoming messages. When an application makes a synchronous method call, OLE enters a modal wait loop that handles the required replies and dispatches incoming messages to applications capable of processing them.

OLE manages method calls by assigning an identifier called a *logical thread ID*. A new one is assigned when a user selects a menu command or when the application initiates a new OLE operation. Subsequent calls that relate to the initial OLE call are assigned the same logical thread ID as the initial call.

CHAPTER 4

Registering Object Applications

Most OLE registry information is stored in subkeys and named values under the following registery keys:

HKEY_LOCAL_MACHINE\SOFTWARE\Classes

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE

To obtain general information about registering an object application or about an OLE registry entry make a selection from one of the topics listed below.

OLE Registry Entries

The topics in this section contain general information about registering OLE applications.

- Installation and Setup
- OLE Registry Functions
- Registering OLE 2 Libraries
- Checking Registration During Run Time
- Specifying Unknown User Types
- Conventions Used in Examples

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID

The topics under the AppID key describe registry entries related to distributed COM.

- RemoteServerName
- ActivateAtStorage
- LocalService
- ServiceParameters
- RunAs
- LaunchPermission
- AccessPermission

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID Key

The topics under the CLSID key contain information that is used by the default OLE handler to return information about a class when it is in the running state.

- AppID
- AutoConvertTo
- AutoTreatAs
- AuxUserType
- *<CLSID>*
- Conversion
- DataFormats
- DefaultIcon
- InprocHandler
- InprocHandler32
- InprocServer

InprocServer32
Insertable
Interface
LocalServer
LocalServer32
MiscStatus
ProgID
ToolBoxBitmap32
TreatAs
Verb
Version

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\<*FileExtension*>

The file extension key associates a file extension with a ProgID, indicating that OLE 2 can handle requests fromthe Windows 3.1 File Manger. It is also used by File Monikers and by **GetClassFile** to supply the associated CLSID.

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\FileType

The **FileType** key is used by **GetClassFile** on non-compound files to obtain a CLSID.

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface

The **Interface** key registers new interfaces with OLE by mapping an interface ID with a CLSID.

BaseInterface
NumMethods
ProxyStubClsid
ProxyStubClsid32

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\<*ProgID*>

The ProgID key maps a component's CLSID to a ProgID.

CLSID
Insertable
Shell

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\<*VersionIndependentProgID*>

The version independent ProgID key associates a ProgID with a CLSID and is used to determine the latest version of an object application.

CLSID
CurVer

HKEY_LOCAL_MACHINE\Software|Microsoft\OLE

The values under this key control Distributed COMs call-level security features in application that do not call **CoInitializeSecurity**.

EnableDCOM
DefaultLaunchPermission

DefaultAccessPermission
LegacyAuthenticationLevel
LegacyImpersonationLevel
LegacyMutualAuthentication
LegacySecureReferences

CHAPTER 5

Error Handling

At the source code level, all error values consist of three parts, separated by underscores. The first part is the prefix that identifies the facility associated with the error, the second part is E for error, and the third part is a string that describes the actual condition. For example, STG_E_MEDIUMFULL is returned when there is no space left on a hard disk. The STG prefix indicates the storage facility, the E indicates that the status code represents an error, and the MEDIUMFULL provides specific information about the error. Many of the values that you might want to return from an interface method or function are defined in winerror.h.

Success, warning, and error values are returned using a 32-bit number known as a result handle, or HRESULT. HRESULTs are really not handles to anything, but merely 32-bit values with several fields encoded in the value. A zero result indicates success and a non-zero result indicates failure.

HRESULTs work differently depending on the platform you are using. On 16-bit platforms, an HRESULT is generated from a 32-bit value known as a status code, or SCODE. On 32-bit platforms, an HRESULT is the same as an SCODE; they are synonymous. In fact, the SCODE is no longer used. 32-bit OLE uses only HRESULTs.

For more information on errors, see the following sections:

- **Structure of OLE Error Codes**
- **Codes in FACILITY_ITF**
- **Using Macros for Error Handling**
- **Error Handling Strategies**
- **Handling Error Information**

Structure of OLE Error Codes

SCODES on 16-bit platforms are divided into four fields: a severity code, a context field, a facility field, and an error code. The format of an SCODE on a 16-bit platform is shown below; the numbers indicate bit positions.

{ewc msdncd, EWGraphic, bsd23517 0 /a "SDK.BMP"}

HRESULTs on 32-bit platforms have the following format.

{ewc msdncd, EWGraphic, bsd23517 1 /a "SDK.BMP"}

The severity code in the 16-bit SCODE and the high order bit in the HRESULT indicates whether the return value represents success or failure. If set to zero, SEVERITY_SUCCESS, the value indicates success. If set to 1, SEVERITY_ERROR, it indicates failure.

In the 16-bit version of the SCODE, the context field is reserved; this field does not exist in the 32-bit version. The R, C, N, and r bits are reserved in both.

The facility field in both versions indicates the area of responsibility for the error. There are currently five facilities: FACILITY_NULL, FACILITY_ITF, FACILITY_DISPATCH, FACILITY_RPC, and FACILITY_STORAGE. If new facilities are necessary, Microsoft allocates them because they need to be unique. Most SCODEs and HRESULTs set the facility field to FACILITY_ITF, indicating an interface method error. The following table describes the various facility fields.

Facility	Description
FACILITY_NULL	For broadly applicable common status codes such as S_OK. This facility code has a value of zero.
FACILITY_ITF	For most status codes returned from interface methods, value is defined by the interface. That is, two SCODEs or HRESULTs with exactly the same 32-bit value returned from two different interfaces might have different meanings. This facility has a value of 4.
FACILITY_DISPATCH	For late binding **IDispatch** interface errors. This facility has a value of 2.
FACILITY_RPC	For status codes returned from remote procedure calls. This facility has a value of 1.
FACILITY_STORAGE	For status codes returned from **IStorage** or **IStream** method calls relating to structured storage. Status codes whose code (lower 16 bits) value is in the range of DOS error codes (that is, less than 256) have the same meaning as the corresponding DOS error. This facility has a value of 3.
FACILITY_WINDOWS	Used for additional error codes from Microsoft-defined interfaces.
FACILITY_WIN32	Used to provide a means of handling error codes from functions in the Win32 API as an HRESULT. Error codes in 16-bit OLE that duplicated Win32 error codes have also been changed to FACILITY_WIN32.

The code field is a unique number that is assigned to represent the error or warning.

By convention, **HRESULTs** generally have names in the following form:

```
<Facility>_<Severity>_<Reason>
```

where <*Facility*> is either the facility name or some other distinguishing identifier, <*Severity*> is a single letter, S or E, that indicates the severity of the error, and <*Reason*> is an identifier that describes the meaning of the code. For example, the status code STG_E_FILENOTFOUND indicates a storage-related error has occurred; specifically, a requested file does not exist. Status codes from FACILITY_NULL omit the <*Facility*>_ prefix.

Error codes are defined within the context of an interface implementation. Once defined, success codes cannot be changed or new success codes added. However, new failure codes can be written since they generally only provide hints at what might have gone wrong. Microsoft reserves the right to define new failure codes (but not success codes) for the interfaces described in this book in FACILITY_ITF or in new facilities.

Codes in FACILITY_ITF

These **HRESULTSs** with facilities such as FACILITY_NULL and FACILITY_RPC have universal meaning because they are defined at a single source: Microsoft. However, **HRESULTs** in FACILITY_ITF are determined by the interface method (or function) from which they are returned. That is, the same 32-bit value in FACILITY_ITF returned from two different interface methods might have different meanings.

The reason **HRESULTs** in FACILITY_ITF can have different meanings in different interfaces is that **HRESULTs** are kept to an efficient data type size of 32 bits. Unfortunately 32 bits is not large enough for the development of an allocation system for error codes that avoid conflict between codes allocated by different non-communicating programmers at different times in different places (unlike the handling of interface IDs and CLSIDs). As a result, the 32-bit **HRESULT** is structured in a way that Microsoft can define some universally-defined error codes, while allowing other programmers to define new error codes without fear of conflict. The status code convention is as follows:

1. Status codes in facilities *other than* FACILITY_ITF can only be defined by Microsoft.
2. Status codes in facility FACILITY_ITF are defined solely by the developer of the interface or API that returns the status code. To avoid conflicting error codes, whoever defines the interface is responsible for coordinating and publishing the FACILITY_ITF status codes associated with that interface.

All the OLE-defined FACILITY_ITF codes have a code value in the range of 0x0000 - 0x01FF. While it is legal to use any codes in FACILITY_ITF, it is recommended that only code values in the range of 0x0200 – 0xFFFF be used. This recommendation is made as a means of reducing confusion with any OLE-defined errors.

It is also recommended that developers define new functions and interfaces to return error codes as defined by OLE and in facilities other than FACILITY_ITF. In particular, interfaces that have any chance of being remoted using RPC in the future should define the FACILITY_RPC codes as legal. E_UNEXPECTED is a specific error code that most developers will want to make universally legal.

Using Macros for Error Handling

OLE defines a number of macros that make it easier to work with SCODEs on 16-bit platforms and HRESULTs on both platforms. Some of the macros and functions below provide conversion between the two data types and are quite useful in code that runs only on 16-bit platforms, code that runs on both 16-bit and 32-bit platforms, and 16-bit code that is being ported to a 32-bit platform. These same macros are meaningless in 32-bit environments and are available in order to provide compatibility and make porting easier. Newly written code should use the **HRESULT** macros and functions.

Error handling macros are listed below. Refer to *The OLE Programmer's Reference* for a complete description of each.

Macro	Description
GetScode	(Obsolete) Returns an SCODE given an HRESULT.
ResultFromScode	(Obsolete) Returns an HRESULT given an SCODE.
PropagateResult	(Obsolete) Generates an HRESULT to return to a function in cases where an error is being returned from an internally called function.
MAKE_HRESULT	Returns an HRESULT given an SCODE that represents an error.
MAKE_SCODE	Returns an SCODE given an HRESULT.
HRESULT_CODE	Extracts the error code part of the HRESULT.
HRESULT_FACILITY	Extracts the facility from the HRESULT.
HRESULT_SEVERITY	Extracts the severity bit from the SEVERITY.
SCODE_CODE	Extracts the error code part of the SCODE.
SCODE_FACILITY	Extracts the facility from the SCODE.
SCODE_SEVERITY	Extracts the severity field from the SCODE.
SUCCEEDED	Tests the severity of the SCODE or HRESULT - returns TRUE if the severity is zero and FALSE if it is one.
FAILED	Tests the severity of the SCODE or HRESULT - returns TRUE if the severity is one and FALSE if it is zero.
IS_ERROR	Provides a generic test for errors on any status value.
FACILITY_NT_BIT	Defines bits so macros are guaranteed to work.
RESULT_FROM_WIN32	Maps a Win32 error value into an HRESULT. This assumes that Win32 errors fall in the range -32k to 32k.
HRESULT_FORM_NT	Maps an NT status value into an HRESULT.

Note Calling **MAKE_HRESULT** for S_OK verification carries a performance penalty. You should not routinely use **MAKE_HRESULT** for successful results.

Error Handling Strategies

Because interface methods are virtual, it is not possible to know, as a caller, the full set of values that may be returned from any one call. One implementation of a method may return five values; another may return eight. The *OLE Programmer's Reference* lists common values that may be returned for each method; these are the values that you must check for and handle in your code because they have special meanings. Other values may be returned, but since they are not meaningful, you do not need to write special code to handle them. A simple check for zero or not zero is adequate.

HRESULTs

The values of some **HRESULT**s have been changed in 32-bit OLE to eliminate all duplication and overlapping with Win32 error codes. Those that duplicate Win32 error codes have been changed to FACILITY_WIN32 and those that overlap remain in FACILITY_NULL. Following is a list of their 32-bit values:

HRESULT	Value
E_UNEXPECTED	0x8000FFFF
E_NOTIMPL	0x80004001
E_OUTOFMEMORY	0x8007000E
E_INVALIDARG	0x80070057
E_NOINTERFACE	0x80004002
E_POINTER	0x80004003
E_HANDLE	0x80070006
E_ABORT	0x80004004
E_FAIL	0x80004005
E_ACCESSDENIED	0x80070005

Handling Error Information

Keep in mind that it is legal to return a status code only from the implementation of an interface method sanctioned as legally returnable. Failure to observe this rule invites the possibility of conflict between returned error code values and those sanctioned by the application. In particular, pay attention to this potential problem when propagating error codes from internally-called functions.

Applications that call interfaces should treat any unknown returned error code (as opposed to a success code) as synonymous with E_UNEXPECTED. This practice of handling unknown error codes is required by clients of the OLE-defined interfaces and APIs. Because typical programming practice is to handle a few specific error codes in detail and treat the rest generically, this requirement of handling unexpected or unknown error codes is easily met.

The following code sample shows the recommended way of handling unknown errors:

```
HRESULT hrErr;
hrErr = xxMethod();

switch (GetScode(hrErr))  {
    case NOERROR:
      //success
      break;

    case x1:
      .
      .
      break;

    case x2:
      .
      .
      break;

    case E_UNEXPECTED:
    default:
      //general failure
      break;
}
```

The following error check is often used with those routines that don't return anything special (other than S_OK or some unexpected error):

```
if (xxMethod() == NOERROR)
    //success
else
    //general failure;
```

Monikers

A moniker in COM is not only, as the name implies, a way to identify an object. A moniker is also implemented as an object. This object provides services allowing a component to obtain a pointer to the object identified by the moniker. This process is referred to as binding.

Monikers are objects that implement the IMoniker_com_IMoniker interface, and are generally implemented in DLLs as component objects. There are two ways of viewing the use of monikers: as a moniker client, a component that uses a moniker to get a pointer to another object, and as a moniker provider, a component that supplies monikers identifying its objects to moniker clients.

OLE uses monikers to connect to and activate objects, whether they are in the same machine or across a network. A very important use is for network connections. They are also used to identify, connect to, and run OLE Compound Document link objects. In this case, the link source acts as the moniker provider, and the container holding the link object acts as the moniker client.

This chapter describes the following:

For basic COM information, refer to **The Component Object Model**.

Moniker Clients

Moniker clients must start by getting a moniker. There are several ways for a moniker client to get a moniker. For example, in OLE Compound Documents, when the end-user creates a linked item (either using Insert Object dialog, the clipboard, or drag-and drop), a moniker is embedded as part of the linked item. In that case, the programmer has minimal contact with monikers. Programmatically, if you have an interface pointer to an object that implements the **IMoniker** interface, you can use that to get a moniker, and there are methods on other interfaces that are defined to return monikers.

There are different kinds of monikers, which are used to identify different kinds of objects, but to a moniker client, all monikers look the same. A moniker client simply calls **IMoniker::BindToObject** on a moniker and gets an interface pointer to the object that the moniker identifies. No matter whether the moniker identifies an object as large as an entire spreadsheet or as small as a single cell within a spreadsheet, calling **IMoniker::BindToObject** will return a pointer to that object. If the object is already running, **IMoniker::BindToObject** will find it in memory. If the object is stored passively on disk, **IMoniker::BindToObject** will locate a server for that object, run the server, and have the server bring the object into the running state. All the details of the binding process are hidden from the moniker client. Thus, for a moniker client, using the moniker is very simple.

Moniker Providers

In general, a component should be a moniker provider when it allows access to one of its objects, while still controlling the object's storage. If a component is going to hand out monikers that identify its objects, it must be capable of performing the following tasks:

- On request, create a moniker that identifies an object.
- Enable the moniker to be bound when a client calls **IMoniker::BindToObject** on it.

A moniker provider must create a moniker of an appropriate *moniker class* to identify an object. The moniker class refers to a specific implementation of the **IMoniker** interface that defines the type of moniker created. While you can implement **IMoniker** to create a new moniker class, it is frequently unnecessary because OLE provides implementations of several different moniker classes, each with its own CLSID. Refer to **OLE Moniker Implementations** for descriptions of moniker classes that OLE provides.

OLE Moniker Implementations

OLE provides implementations of several monikers for different situations:

- **File Monikers**
- **Composite Monikers**
- **Item Monikers**
- **Anti-monikers**
- **Pointer monikers**
- **Class Monikers**

The file, composite, and item monikers are the most frequently used monikers, as they can be used to make nearly any object in any location. Anti- and pointer monikers are primarily used inside OLE, but have some application in implementing custom monikers.

File Monikers

File monikers are the simplest moniker class. File monikers can be used to identify any object that is stored in its own file. A file moniker acts as a wrapper for the path name the native file system assigns to the file. Calling **IMoniker::BindToObject** for this moniker would cause this object to be activated and then would return an interface pointer to the object. The source of the object named by the moniker must provide an implementation of the **IPersistFile** interface to support binding a file moniker. File monikers can represent either a complete or a relative path.

For example, the file moniker for a spreadsheet object stored as the file C:\WORK\MYSHEET.XLS would contain information equivalent to that path name. The moniker would not necessarily consist of the same string, however. The string is just its *display name*, a representation of the moniker's contents that is meaningful to an end user. The display name, which is available through the **IMoniker::GetDisplayName** method, is used only when displaying a moniker to an end-user. This method gets the display name for any of the moniker classes. Internally, the moniker may store the same information in a format that is more efficient for performing moniker operations, but isn't meaningful to users. Then, when this same object is bound through a call to the **BindToObject** method. the object would be activated, probably by loading the file into the spreadsheet.

OLE offers moniker providers the helper API **CreateFileMoniker** that creates a file moniker object and returns its pointer to the provider.

Composite Monikers

One of the most useful features of monikers is that you can concatenate or *compose* monikers together. A composite moniker is a moniker that is a composition of other monikers, and can determine the relation between the parts. This lets you assemble the complete path to an object given two or more monikers that are the equivalent of partial paths. You can compose monikers of the same class (like two file monikers) or of different classes (like a file moniker and an item moniker). If you were to write your own moniker class, you could also compose your monikers with file or item monikers. The basic advantage of a composite is that it gives you one piece of code to implement every possible moniker that is a combination of simpler monikers. That reduces tremendously the need for specific custom moniker classes.

Because monikers of different classes can be composed with one another, monikers provide the ability to join multiple namespaces. The file system defines a common namespace for objects stored as files because all applications understand a file-system path name. Similarly, a container object also defines a private namespace for the objects that it contains, because no container understands the names generated by another container. Monikers allow these name spaces to be joined because file monikers and item monikers can be composed. A moniker client can search the namespace for *all objects* using a single mechanism. The client simply calls **IMoniker::BindToObject** on the moniker, and the moniker code handles the rest. A call to **IMoniker::GetDisplayName** on a composite creates a name using the concatenation of all the individual monikers' display names.

Furthermore, because you can write your own moniker class, moniker composition allows you to add customized extensions to the namespace for objects.

Sometimes two monikers of specific classes can be combined in a special way. For example, a file moniker representing an incomplete path and another file moniker representing a relative path can be combined to form a single file moniker representing the complete path. For example, the file monikers **c:\work\art** could be composed with the relative file moniker **..\backup\myfile.doc** to equal **c:\work\backup\myfile.doc**. This is an example of "non-generic" composition.

"Generic" composition, on the other hand, permits the connection of any two monikers, no matter what their classes. For example, you could compose an item moniker onto a file moniker, though not, of course, the other way around.

Because a non-generic composition depends on the class of the monikers involved, its details are defined by a the implementation of a particular moniker class. You can define new types of non-generic compositions if you write a new moniker class. By contrast, generic compositions are defined by OLE. Monikers created as a result of generic composition are called generic composite monikers.

These three classes – file monikers, item monikers, and generic composite monikers – all work together, and they are the most commonly used classes of monikers.

Moniker clients should call **IMoniker::ComposeWith** to create a composite on moniker with another. The moniker it is called on internally decides whether it can do a generic or non-generic composition. If the moniker implementation determines that a generic composition is usable, OLE provides the **CreateGenericComposite** API function to facilitate this.

Item Monikers

Another OLE-implemented moniker class is the item moniker, which can be used to identify an object contained in another object. One type of contained object is an OLE object embedded in a compound document. A compound document could identify the embedded objects it contains by assigning each one an arbitrary name, such as "embedobj1," "embedobj2," and so forth. Another type of contained object is a user selection in a document, such as a range of cells in a spreadsheet or a range of characters in a text document. An object that consists of a selection is called a *pseudo-object* because it isn't treated as a distinct object until a user marks the selection. A spreadsheet might identify a cell range using a name such as "1A:7F," while a word processing document might identify a range of characters using the name of a bookmark.

An item moniker is useful primarily when concatenated – or "composed" – with another moniker, one that identifies the container. An item moniker is usually created, then composed onto (for example) a file moniker to create the equivalent of a complete path to the object. For example, you can compose the file moniker "C:\work\report.doc" (which identifies the container object) with the item moniker "embedobj1" (which identifies an object within the container) to form the moniker "C:\work\report.doc\embedobj1," which uniquely identifies a particular object within a particular file. You can also concatenate additional item monikers to identify deeply nested objects. For example, if "embedobj1" is the name of a spreadsheet object, then to identify a certain range of cells in that spreadsheet object you could append another item moniker to create a moniker that would be the equivalent of "C:\work\report.doc\ embedobj1\1A:7F."

When combined with a file moniker, an item moniker forms a complete path. Item monikers thus extend the notion of path names beyond the file system, defining path names to identify individual objects, not just files.

There is a significant difference between an item moniker and a file moniker. The path contained in a file moniker is meaningful to anyone who understands the file system, while the partial path contained in an item moniker is meaningful only to *a particular container*. Everyone knows what "c:\work\report.doc" refers to, but only one particular container object knows what "1A:7F" refers to. One container cannot interpret an item moniker created by another application; the only container that knows which object is referred to by an item moniker is the container that assigned the item moniker to the object in the first place. For this reason, the source of the object named by the combination of a file and item moniker must not only implement **IPersistFile** to facilitate binding the file moniker, but also **IOleItemContainer** to facilitate resolving the name of the item moniker into the appropriate object, in the context of a file.

The advantage of monikers is that someone using a moniker to locate an object doesn't need to understand the name contained within the item moniker, as long as the item moniker is part of a composite. Generally, it would not make sense for an item moniker to exist on its own. Instead, you would compose an item moniker onto a file moniker. You would then call **IMoniker::BindToObject** on the composite, which binds the individual monikers within it, interpreting the names.

To create an item moniker object and return its pointer to the moniker provider, OLE provides the helper API **CreateItemMoniker**. This function creates an item moniker object and returns its pointer to the provider.

Anti-monikers

OLE provides an implementation of a special type of moniker called an *anti-moniker*. You use this moniker in the creation of new moniker classes. You use it as the inverse of the moniker that it is composed onto, effectively canceling that moniker, in much the same way that the ".." operator moves up a directory level in a file system command.

It is necessary to have an anti-moniker available, because once a composite moniker is created, it is not possible to delete parts of the moniker if, for example, an object moves. Instead, you use an anti-moniker to remove one or more entries from a composite moniker.

Anti-monikers are a moniker class explicitly intended for use as an inverse. COM defines an API function named **CreateAntiMoniker**, which returns an anti-moniker. You generally use this function to implement the **IMoniker::Inverse** method.

An anti-moniker is *only* an inverse for those types of monikers that are implemented to treat anti-monikers as an inverse. For example, if you want to remove the last piece of a composite moniker, you should not create an anti-moniker and compose it to the end of the composite. You cannot be sure that the last piece of the composite considers an anti-moniker to be its inverse. Instead, you should call **IMoniker::Enum** on the composite moniker, specifying **FALSE** as the first parameter. This creates an enumerator that returns the component monikers in reverse order. Use the enumerator to retrieve the last piece of the composite, and call **IMoniker::Inverse** on that moniker. The moniker returned by **IMoniker::Inverse** is what you need to remove the last piece of the composite.

Pointer Monikers

A pointer moniker identifies an object that can exist only in the active or running state. This differs from other classes of monikers, which identify objects that can exist either in the passive or active state.

Suppose, for example, an application has an object that has no persistent representation. Normally, if a client of your application needs access to that object, you could simply pass the client a pointer to the object. However, suppose your client is expecting a moniker. The object cannot be identified with a file moniker, since it isn't stored in a file, nor with an item moniker, since it isn't contained in another object.

Instead, your application can create a pointer moniker, which is a moniker that simply contains a pointer internally, and pass that to the client. The client can treat this moniker like any other. However, when the client calls **IMoniker::BindToObject** on the pointer moniker, the moniker code does not check the Running Object Table (ROT) or load anything from storage. Instead, the moniker code simply calls **IUnknown::QueryInterface** on the pointer stored inside the moniker.

Pointer monikers allow objects that exist only in the active or running state to participate in moniker operations and be used by moniker clients. One important difference between pointer monikers and other classes of monikers is that pointer monikers cannot be saved to persistent storage. If you do, calling the **IMoniker::Save** method returns an error. This means that pointer monikers are useful only in specialized situations. You can use the **CreatePointerMoniker** API function if you need to use a pointer moniker.

Class Monikers

Although classes are typically identified directly with CLSID's to APIs such as **CoCreateInstance** or **CoGetClassObject**, classes may also now be identified with a moniker called a class moniker. Class monikers bind to the class object of the class for which they are created.

The ability to identify classes with a moniker supports useful operations which are otherwise unweildy. For example, file monikers traditionally only supported rich binding to the class associated with the class of file they referred to - a moniker to an Excel file would bind to an instance of an Excel object, and a moniker to a GIF image would bind to an instance of the currently registered GIF handler. Class moniker allows you to indicate the class you want to use to manipulate a file through composition with a file moniker. A class moniker for a 3D charting class composed with a moniker to an Excel file yields a moniker which binds to an instance of the 3D charting object and initializes the object with the contents of the Excel file.

Class monikers are therefore most useful in composition with other types of monikers, such as file monikers or item monikers.

Class monikers may also be composed to the right of monikers supporting binding to the **IClassActivator** interface. When composed in this manner **IClassActivator** simply gives access to the class object and instances of the class through **IClassActivator::GetClassObject**. Class monikers may be identified through **IMoniker::IsSystemMoniker** which returns MKSYS_CLASSMONIKER in *pdwMkSys*.

Programmers typically create class monikers using the **CreateClassMoniker** function or through **MkParseDisplayName** (see **IMoniker - Class Moniker Implementation** about **ParseDisplayName** for details).

Asynchronous Monikers

The OLE moniker architecture provides a consistent, extensible programming model for working with Internet objects, providing methods for parsing names, representing Universal Resource Locators (URLs) as printable names, and locating and binding to the objects represented by URL strings. (Also see **URL Monikers**.) Standard OLE monikers (notably, item, file, and pointer monikers), however, are inappropriate for the Internet because they are synchronous, returning a pointer to an object or its storage only at such time as all data is available. Depending on the amount of data to be downloaded, binding synchronously can tie up the client's user interface for prolonged periods.

The Internet requires new approaches to application design. Applications should be able to perform all expensive network operations asynchronously to avoid stalling the user interface. An application should be able to trigger an operation and receive notification on full or partial completion. At that point, the application should have the choice either of proceeding with the next step of the operation or providing additional information as needed. As a download proceeds, an application should also be able to provide users with progress information and the opportunity to cancel the operation at any time.

Asynchronous monikers provide these capabilities, as well as various levels of asynchronous binding behavior, while providing backwards compatibility for applications that are either unaware of, or do not require asynchronous behavior. Another OLE technology, **Asynchronous Storage**, works with asynchronous monikers to provide asynchronous downloading of an Internet object's persistent state. The asynchronous moniker triggers the bind operation and sets up the necessary components, including storage and stream objects, byte-array objects, and notification sinks. Once the components are connected, the moniker gets out of the way and the rest of the bind is executed mainly between the components implementing the asynchronous storage components and the object.

Asynchronous Versus Synchronous Monikers

A client of a stadard, synchronous OLE moniker typically creates and holds a reference to the moniker, as well as the bind-context to be used during binding. The components involved in using traditional monikers are shown in the following diagram.

{ewc msdncd, EWGraphic, bsd23519 0 /a "SDK_MON1.WMF"}

Clients typically create standard monikers by calling APIs functions such as **CreateFileMoniker**, **CreateItemMoniker**, or **CreatePointerMoniker** or, because they are can be saved to persistent storage, through **OleSaveToStream** and **OleLoadFromStream**. Monikers may also be obtained from a container object by calling **IBindHost::CreateMoniker**. Clients create bind contexts by calling the **CreateBindCtx** API function, then pass the bind context to the moniker with calls to **IMoniker::BindToStorage** or **IMoniker::BindToObject**.

As shown in the following diagram, a client of an asynchronous moniker also creates and holds a reference to the moniker and bind-context to be used during binding.

{ewc msdncd, EWGraphic, bsd23519 1 /a "SDK_MON2.WMF"}

In order to get asynchronous behavior, the client implements the **IBindStatusCallback** interface on a bind-status-callback object and calls the either the **RegisterBindStatusCallback** or **CreateAsyncBindCtx** API functions to register this interface with the bind-context. The moniker passes a pointer to its **IBinding** interface in a call to **IBindStatusCallback::OnStartBinding**. The client tells the asynchronous moniker *how* it wants to bind on return from the moniker's call to **IBindStatusCallback::GetBindInfo**.

Asynchronous Versus Synchronous Binding

The client may check to see if the moniker is asynchronous by calling the **IsAsyncMoniker** API function. If the client returns the BINDF_ASYNCHRONOUS flag, rather than returning an object pointer or a storage pointer from subsequent calls to **IMoniker::BindToStorage** or **IMoniker::BindToObject**, the moniker returns MK_S_ASYNCHRONOUS in place of the object pointer and NULL in place of the storage pointer. In response, the client should wait to receive the requested object or storage during **IBindStatusCallback::OnDataAvailable** and **IBindStatusCallBack::OnObjectAvailable**.

The callback object also receives progress notification through **IBindStatusCallback::OnProgress**, data availability notification through **IBindStatusCallback::OnDataAvailable**, and various other notifications from the moniker about the status of the binding operation.

If the client does not return the BINDF_ASYNCHRONOUS flag from the moniker's call to **GetBindInfo**, the bind operation will proceed synchronously, and the desired object or storage will be returned from subsequent calls to **IMoniker::BindToObject** or **IMoniker::BindToStorage**. Similarly, if the client desires synchronous operation and does not wish to receive *any* progress notifications or callbacks, it can request an asynchronous moniker to behave synchronously by not implementing **IBindStatusCallback**. In such cases, the asynchronous moniker will behave like a standard synchronous moniker.

Asynchronous Versus Synchronous Storage

Asynchronous monikers may also return an **Asynchronous Storage** object in the **IBindStatusCallback::OnDataAvailable** notification. This storage object may allow access to *some* of the object's persistent data while the binding is still in progress. A client can choose between two modes for the storage: blocking and non-blocking. In blocking mode, which is compatible with current implementations of storage objects, if data is unavailable, the call blocks until data arrives. In nonblocking mode, rather than blocking the call, the storage object returns a new error E_PENDING when data is unavailable. An client aware of asynchronous binding and storage notes this error and waits for further notifications (**IBindStatusCallback::OnDataAvailable**) to retry the operation. A client can choose between a synchronous (blocking) and asynchronous (non-blocking) storage by choosing whether or not to set the BINDF_ASYNCSTORAGE flag in the *pgrfBINDF* value returned to **IBindStatusCallback::GetBindInfo**.

Data-Pull Model Versus Data-Push Model

A client of an asynchronous moniker can choose between a data-pull and data-push model for driving an asynchronous **IMoniker::BindToStorage** operation and receiving asynchronous notifications. In the data-pull model, the client drives the bind operation, and the moniker only provides data to the client as it is read. In other words, after the first call to **IBindStatusCallback::OnDataAvailable**, the moniker does not provide any data to the client unless the client has consumed all of the data that is already available.

Because data is only downloaded as it is requested, clients that choose the data-pull model must make sure to read this data in a timely manner. In the case of Internet-downloads with **URL Monikers**, the bind operation may fail if a client waits too long before requesting more data.

In the data-push model, the moniker drives the asynchronous bind operation and continuously notifies the client whenever new data is available. The client may choose whether or not to read the data at any point during the bind operation, but the moniker will drive the bind operation to completion regardless.

URL Monikers

The OLE moniker architecture provides a convenient programming model for working with URLs. The moniker architecture supports extensible and complete name parsing through the **MkParseDisplayName**(**Ex**) function and the **IParseDisplayName** and **IMoniker** interfaces, as well as printable names through the **IMoniker::GetDisplayName** method. The **IMoniker** interface is the way you actually use URLs you encounter, and building components that fit into the moniker architecture is the way to actually extend URL namespaces in practice.

A new system-provided moniker class, the URL Moniker, provides a framework for building and using certain URLs. Since URLs frequently refer to resources across high-latency networks, the URL Moniker supports asynchronous as well as synchronous binding. The URL Moniker does not currently support **Asynchronous Storage**.

The following diagram shows the components involved in using URL Monikers. All these components should be familiar from the discussion of asynchronous monikers in the preceding section of this chapter.

{ewc msdncd, EWGraphic, bsd23519 2 /a "SDK_MON3.WMF"}

Like all moniker clients, a user of URL Monikers typically creates and holds a reference to the moniker as well as to the bind-context to be used during binding (**IMoniker::BindToStorage** or **IMoniker::BindToObject**). To support asynchronous binding, the client can implement a bind-status-callback object, which implements the **IBindStatusCallback** interface, and register it with the bind-context using the **RegisterBindStatusCallback** API function. This object will receive the transport's **IBinding** interface during calls to **IBindStatusCallback::OnStartBinding**.

The URL Moniker identifies the protocol being used by parsing the URL prefix, then retrieves the **IBinding** interface from the transport layer. The client uses **IBinding** to support pausing, cancellation, and prioritization of the binding operation. The callback object also receives progress notification through **IBindStatusCallback::OnProgress**, data availability notification through **IBindStatusCallback::OnDataAvailable**, and various, other, transport-layer notifications about the status of the binding. The URL Moniker or specific transport layers may also request extended information from the client via **IBindStatusCallback::QueryInterface**, allowing the client to provide protocol-specific information that will affect the bind operation.

Callback Synchronization

The asynchronous WinInet API (used for the most common protocols) leaves the synchronization of the callback mechanism and the calling application as an exercise for the client. This is intentional because it allows the greatest degree of flexibility. The default protocols and the URL Moniker implementation perform this synchronization and guarantee that single- and apartment-threaded applications never have to deal with free-thread-style contention. That is, the client's **IEnumFORMATETC** and **IBindStatusCallback** interfaces are called only on their proper threads. This feature is transparent to the user of the URL Moniker as long each thread that calls **IMoniker::BindToStorage** and **IMoniker::BindToObject** has a message queue.

The Asynchronous Moniker specification requires more precise control over the prioritization and management of downloads than is allowed for either by WinSock or WinInet. Accordingly, a URL Moniker manages all the downloads for any given caller's thread, using—as part of its synchronization—a priority scheme based on the **IBinding** specification.

Media-Type Negotiation

Many application-layer Internet protocols are based on the exchange of messages in a simple, flexible format called Multipurpose Internet Mail Extensions (MIME). Although MIME originated as a standard for exchanging electronic mail messages, it is used today by a wide variety of applications to specify mutually understood data formats as MIME , or media, types. The process is called media-type negotiation.

Media types are simple strings that denote a type and subtype (such as "text/plain" or "text/HTML"). They are used to label data or qualify a request. A Web browser, for example, as part of an HTTP request-for-data or request-for-info, specifies that it is requesting "image/gif" or "image/jpeg" Media Types, to which a Web server responds by returning the appropriate media type and, if the call was a request-for-data, the data itself in the requested format.

Media-type negotiation is often similar to how existing desktop applications negotiate with the system clipboard to determine which data format to paste when a user chooses Edit/Paste or queries for formats when receiving an **IDataObject** pointer during a drag-and-drop operation. The subtle difference in HTTP media-type negotiation is that the client does not know ahead of time which formats the server has available. Therefore, the client specifies up-front the media types it supports, in order of greatest fidelity, and the server responds with the best available format.

URL Monikers support media-type negotiation as a way for Internet clients and servers to agree upon formats to be used when downloading data in **BindToStorage** operations. To support media-type negotiation, a client implements the **IEnumFORMATETC** interface and calls the **RegisterFormatEnumerator** API function to register it with the bind context. The format enumerator lists the formats the client can accept. A URL Moniker translates these formats into media types when binding to HTTP URLs.

The possible media types requested by the client are represented to URL Monikers through **FORMATETC** structures available from the **IEnumFORMATETC** enumerator registered by the caller on the bind-context: Each **FORMATETC** specifies a clipboard format identifying the media type. For example, the following code fragment specifies that the media type is PostScript®.

```
FORMATETC fmtetc;
fmtetc.cfFormat = RegisterClipboardFormat(CF_MIME_POSTSCRIPT);
. . .
```

(For more information on the **FORMATETC** structure, see the *OLE Programmer's Reference* in the Win32 SDK.)

A client can set the clipboard format to the special media type CF_NULL to indicate that the default media type of the resource pointed to by the URL should be retrieved. This format is usually the last one in which the client is interested. When no enumerator is registered with the bind context, a URL Moniker works as if an enumerator containing a single **FORMATETC** with *cfFormat*=CF_NULL is available, automatically downloading the default media-type.

Regardless which media type is to be used, the client is notified of the choice by means of the *pformatetc* argument on its **IBindStatusCallback::OnDataAvailable** method. The callback occurs within the context of the client's call to **IMoniker::BindToStorage**.

Note If received content is of an unrecognized media type, the client automatically calls **RegisterMediaTypes** to register the new type.

URL Moniker API Functions

URL Moniker API Functions insulate developers from the complexities of creating, managing, and using URL Monikers. These monikers, which are fully described in the OLE Programmer's Reference for the ActiveX Development Kit, are as follows:**CreateURLMoniker**, **IsValidURL**, **RegisterMediaTypes**, **CreateFormatEnumerator**, **RegisterFormatEnumerator**, **RevokeFormatEnumerator**, **RegisterMediaTypeClass**, **FindMediaTypeClass**, **GetClassFileOrMime**, **UrlMkSetSessionsOption**.

Connectable Objects

COM connectable objects provide outgoing interfaces to their clients in addition to their incoming interfaces. As a result, objects and their clients can engage in bi-directional communication. Incoming interfaces are implemented on an object and receive calls from external clients of an object while outgoing interfaces are implemented on the client's sink and receive calls from the object. The object defines an interface it would like to use, and the client implements it.

An object defines its incoming interfaces and provides implementations of these interfaces. Incoming interfaces are available to clients through the object's **IUnknown::QueryInterface** method. Clients call the methods of an incoming interface on the object, and the object performs desired actions on behalf of the client.

Outgoing interfaces are also defined by an object, but the client provides the implementations of the outgoing interfaces on a sink object that the client creates. The object then calls methods of the outgoing interface on the sink object to notify the client of changes in the object, to trigger events in the client, or to request something from the client, or, in fact, for any purpose the object creator comes up with.

An example of an outgoing interface is an **IButtonSink** interface defined by a push button control to notify its clients of its events. For example, the button object calls **IButtonSink::OnClick** on the client's sink object when the user clicks the button on the screen. The button control defines the outgoing interface. For a client of the button to handle the event, the client must implement that outgoing interface on a sink object then connect that sink to the button control. Then, when events occur in the button, the button will call the sink at which time the client can execute whatever action it wishes to assign to that button click.

Connectable objects provide a general mechanism for object-to-client communication. Any object that wishes to expose events or notifications of any kind can use this technology. In addition to the general connectable object technology, COM provides many special purpose sink and site interfaces used by objects to notify clients of specific events of interest to the client. For example, **IAdviseSink** may be used by objects to notify clients of data and view changes in the object.

Architecture of Connectable Objects

The connectable object itself is only one piece of the overall architecture of connectable objects. This technology includes:

Connectable object
Implements the **IConnectionPointContainer** interface; creates at least one connection point object; defines an outgoing interface for the client.

Client
Queries the object for **IConnectionPointContainer** to determine if the object is connectable; creates a sink object to implement the outgoing interface defined by the connectable object.

Sink object
Implements the outgoing interface; used to establish a connection to the connectable object.

Connection point object
Implements the **IConnectionPoint** interface and manages connection with the client's sink.

The relationships between client, connectable object, a connection point, and a sink are illustrated in the following diagram:

{ewc msdncd, EWGraphic, bsd23515 0 /a "SDK.WMF"}

Before the connection point object calls methods in the sink interface in step 3, it must **QueryInterface** for the specific interface required, even if the pointer was already passed in the step 2 call to the **Advise** method.

Two enumerator objects are also involved in this architecture though not shown in the illustration. One is created by a method in **IConnectionPointContainer** to enumerate the connection points within the connectable object. The other is created by a method in **IConnectionPoint** to enumerate the connections currently established to that connection point. One connection point can support multiple connected sink interfaces, and it should iterate through the list of connections each time it makes a method call on that interface. This process is known as multi-casting.

When working with connectable objects it is important to understand that the connectable object, each connection point, each sink, and all enumerators are separate objects with separate **IUnknown** implementations, separate reference counts, and separate lifetimes. A client using these objects is always responsible for releasing all reference counts it owns.

Note A connectable object can support more than one client and can support multiple sinks within a client. Likewise, a sink can be connected to more than one connectable object.

The steps for establishing a connection between a client and a connectable object are:

1. The client queries for **IConnectionPointContainer** on the object to determine if the object is connectable. If this call is successful, the client holds a pointer to the **IConnectionPointContainer** interface on the connectable object, and the connectable object reference counter has been incremented. Otherwise, the object is not connectable and does not support outgoing interfaces.
2. If the object is connectable, the client next tries to obtain a pointer to the **IConnectionPoint** interface on a connection point within the connectable object. There are two methods for obtaining this pointer, both in **IConnectionPointContainer** – **FindConnectionPoint** and **EnumConnectionPoints**. There are a few additional steps needed if **EnumConnectionPoints** is used; see below for more information. If successful, the connectable object and the client both support the same outgoing interface. The connectable object defines it and calls it while the client

implements it. The client can then communicate through the connection point within the connectable object.

3. The client then calls **IConnectionPoint::Advise** on the connection point to establish a connection between its sink interface and the object's connection point. After this call, the object's connection point holds a pointer to the outgoing interface on the sink.
4. The code inside **IConnectionPoint::Advise** calls **QueryInterface** on the interface pointer that is passed in, asking for the specific interface identifier to which it connects.
5. The object calls methods on the sink's interface as needed using the pointer held by its connection point.
6. The client calls **IConnectionPoint::Unadvise** to terminate the connection. Then, the client calls **IConnectionPoint::Release** to free its hold on the connection point and, thus, the main connectable object also. The client must also call **IConnectionPointContainer::Release** to free its hold on the main connectable object.

Connectable Object Interfaces

Support for connectable objects requires support for four interfaces:

- **IConnectionPointContainer** on the connectable object
- **IConnectionPoint** on the connection point object
- **IEnumConnectionPoints** on an enumerator object
- **IEnumConnections** on an enumerator object

The latter two are defined as standard enumerators for the types **IConnectionPoint *** and **CONNECTDATA**. See **IEnumXxxx** for more information on enumerators.

Additionally, the connectable object can optionally support **IProvideClassInfo** and **IProvideClassInfo2** to provide enough information to a client so the client can provide support for the outgoing interface at run-time.

Finally, the client must provide a sink object that implements the outgoing interface which is a custom COM interface defined by the connectable object.

The **IConnectionPointContainer**, **IConnectionPoint**, **IProvideClassInfo**, and **IProvideClassInfo2** interfaces are defined as follows:

```
interface IConnectionPointContainer : IUnknown
    {
    HRESULT EnumConnectionPoints([out] IEnumConnectionPoints
        **ppEnum);
    HRESULT FindConnectionPoint([in] REFIID riid
        , [out] IConnectionPoint **ppCP);
    }


interface IConnectionPoint : IUnknown
    {
    HRESULT GetConnectionInterface([out] IID *pIID);
    HRESULT GetConnectionPointContainer([out]
        IConnectionPointContainer **ppCPC);
    HRESULT Advise([in] IUnknown *pUnk, [out] DWORD *pdwCookie);
    HRESULT Unadvise([in] DWORD dwCookie);
    HRESULT EnumConnections([out] IEnumConnections **ppEnum);
    }


interface IProvideClassInfo : IUnknown
    {
    HRESULT GetClassInfo([out] ITypeInfo **ppTI);
    }


interface IProvideClassInfo2 : IProvideClassInfo
    {
    HRESULT GetGUID( );
    }
```

Using IConnectionPointContainer

A connectable object implements **IConnectionPointContainer** (and exposes it through **QueryInterface**) to indicate the existence of outgoing interfaces. For each outgoing interface, the connectable object manages a connection point sub-object which itself implements **IConnectionPoint.** The connectable object therefore contains the connection points, hence the naming of **IConnectionPointContainer** and **IConnectionPoint**.

Through **IConnectionPointContainer**, a client can perform two operations. First, if the client already has the IID for an outgoing interface that it supports, it can locate the corresponding connection point for the IID using **FindConnectionPoint**. The client cannot query for the connection point directly because of the container/contained relationship between the connectable object and its contained connection points. In essence, **FindConnectionPoint** is the **QueryInterface** for outgoing interfaces when the IID is known to the client.

Second, the client can enumerate all connection points within the connectable object through **IConnectionPointContainer::EnumConnectionPoints**. This method returns an **IEnumConnectionPoints** interface pointer for a separate enumerator object. Through **IEnumConnectionPoints::Next** the client can obtain **IConnectionPoint** interface pointers to each connection point.

Once the client obtains the **IConnectionPoint** interface, it must call **IConnectionPoint::GetConnectionInterface** to determine the IID of the outgoing interface supported by each connection point. If the client already supports that outgoing interface, it can establish a connection. Otherwise, it may still be able to support the outgoing interface using information from the connectable object's type library to provide support at run-time. This technique requires that the connectable object support the **IProvideClassInfo** interface as described below.

Note Because the enumerator is a separate object, the client must call **IEnumConnectionPoints::Release** when the enumerator is no longer needed.

In addition, each connection point is an object with a separate reference count from the containing connectable object. Therefore, the client must also call **IConnectionPoint::Release** for each connection point accessed through either the enumerator or through **FindConnectionPoint**.

Using IConnectionPoint

Once the client has a pointer to a connection point, it can then perform several operations as expressed through **IConnectionPoint**. First, **GetConnectionInterface**, as noted above, retrieves the outgoing interface IID supported by the connection point. When used in conjunction with **IEnumConnectionPoints,** this method allows the client to examine the IIDs of all outgoing interfaces supported on the connectable object.

Second, a client can navigate from the connection point back to the connectable object's **IConnectionPointContainer** interface through the **GetConnectionPointContainer** method.

Third, the most interesting methods for the client are **Advise** and **Unadvise**. When a client wishes to connect its own sink object to the connectable object, the client passes the sink's **IUnknown** pointer (or any other interface pointer on the same object) to **Advise**. The connection point queries the sink for the specific outgoing interface that is expected. If that interface is available on the sink, the connection point then stores the interface pointer. From this point until **Unadvise** is called, the connectable object will make calls to the sink through this interface when events occur. To disconnect the sink from the connection point, the client passes a key returned from **Advise** to the **Unadvise** method. **Unadvise** must call **Release** on the sink interface.

Finally, a client can ask a connection point to enumerate all the connections that exist to it through **EnumConnections**. This method creates an enumerator object (with a separate reference count) returning an **IEnumConnections** pointer to it. The client must call **Release** when the enumerator is no longer needed. Additionally, the enumerator returns a series of **CONNECTDATA** structures, one for each connection. Each structure describes one connection using the **IUnknown** pointer of the sink as well as the connection key originally returned from **Advise**. When done with these sink interface pointers, the client must call **IUnknown::Release** on each pointer returned in a **CONNECTDATA** structure.

Using IProvideClassInfo

A connectable object can offer the **IProvideClassInfo[*x*]** interfaces so its clients can easily examine its type information. See Providing Class Information chapter for more information.

This capability is important when dealing with outgoing interfaces, which, by definition, are defined by an object but implemented by a client on its own sink object. In some cases, an outgoing interface is known at compile time to both the connectable object and the sink object; such is the case with **IPropertyNotifySink**.

In other cases, however, only the connectable object knows its outgoing interface definitions at compile time. In these cases, the client must obtain the type information for the outgoing interface so it can dynamically provide a sink supporting the right entry points.

First, as described above, the client can enumerate the connection points and can then call **IConnectionPoint::GetConnectionInterface** for each connection point to obtain the IIDs of outgoing interfaces supported by the connectable object.

Second, the client queries the connectable object for one of the **IProvideClassInfo[*x*]** interfaces. Third, the client calls methods in these interfaces to get the type information for the outgoing interface. Fourth, the client creates a sink object supporting the outgoing interface. Finally, the process continues as described above with the client calling **IConnectionPoint::Advise** to connect its sink to the connection point.

In the type information, the attribute **source** marks an **interface** or **dispinterface** listed under a **coclass** as an outgoing interface. Those listed without this attribute are considered incoming interfaces.

Structured Storage

Traditional file systems face challenges when they try to efficiently store multiple kinds of objects in one document. OLE provides a solution: a file system *within* a file. OLE structured storage defines how to treat a single file entity as a structured collection of two types of objects – storages and streams – that act like directories and files. This scheme is called structured storage. The purpose of structured storage is to reduce the performance penalties and overhead associated with storing separate objects in a flat file

The Evolution of File Systems

Years ago, in the days before disk operating systems, each computer was built to run a single, proprietary application, which had complete and exclusive control of the entire machine. The application would write its persistent data directly to a disk, or drum, by sending commands directly to the disk controller. The application was responsible for managing the absolute locations of data on the disk, making sure that it was not overwriting data that was already there, but since the application was the only one running on the machine, this task was not too difficult.

The advent of computer systems that could run more than one application required some sort of mechanism to make sure that applications did not write over each other's data. Application developers addressed this problem by adopting a single standard for marking which disk sectors were in use and which were free. In time, these standards coalesced to become a disk operating system, which provided various services to the applications, including a file system for managing persistent storage. With the advent of a file system, applications no longer had to deal directly with the physical storage medium. Instead, they simply told the file system to write blocks of data to the disk and let the file system worry about how to do it. In addition, the file system allowed applications to create data hierarchies through the abstraction known as a directory. A directory could contain not only files but other directories. which in turn could contain their own files and directories, and so on.

The file system provided a single level of indirection between applications and the disk, and the result was that every application saw a file as a single contiguous stream of bytes on the disk even though the file system was actually storing the file in discontiguous sectors. The indirection freed the applications from having to care about the absolute position of data on a storage device.

Today, virtually all system APIs for file input and output provide applications with some way to write information into a flat file that applications see as a single stream of bytes that can grow as large as necessary until the disk is full. For a long time these APIs have been sufficient for applications to store their persistent information. Applications have made significant innovations in how they deal with a single stream of information to provide features like incremental "fast" saves.

In a world of component objects, however, storing data in a single flat file is no longer efficient. Just as file systems arose out of the need for multiple applications to share the same storage medium, so now, component objects require a system that allows them to share storage within the conceptual framework of a single file. Even though it is possible to store the separate objects using conventional flat file storage, should one of the objects increase in size, or should you simple add another object, it becomes necessary to load the entire file into memory, insert the new object, and then save the whole file. Such a process can be extremely time-consuming.

The solution provided by OLE is to implement a second level of indirection: a file system *within* a file. Instead of requiring that a large contiguous sequence of bytes on the disk be manipulated through a single file handle with a single seek pointer, OLE structured storage defines how to treat a single file system entity as a structured collection of two types of objects – storages and streams – that act like directories and files.

Storages and Streams

OLE provides a set of services collectively called structured storage. The purpose of these services is to reduce the performance penalties and overhead associated with storing separate objects in a flat file. Instead, OLE stores the separate objects in a single, structured file consisting of two main elements: storage objects and stream objects. Together, they function like a file system within a file.

A storage object is analogous to a file system directory. Just as a directory can contain other directories and files, a storage object can contain other storage objects and stream objects. Also like a directory, a storage object tracks the locations and sizes of the storage objects and stream objects nested beneath it.

A stream object is analogous to the traditional notion of a file. Like a file, a stream contains data stored as a consecutive sequence of bytes.

An OLE compound file consists of a root storage object containing at least one stream object representing its native data along with one or more storage objects corresponding to its linked and embedded objects. The root storage object maps to a filename in whatever file system it happens to reside in. Each of the objects inside the document also is represented by a storage object containing one or more stream objects, and perhaps also containing one or more storage objects. In this way, a document can consist of an unlimited number of nested objects.

Structured storage solves the performance problem because whenever a new object is added to a compound file, or an existing object increases in size, the file does not have to be totally rewritten to storage. Instead, the new data is written to the next available location in permanent storage, and the storage object updates the table of pointers it maintains to track the locations of its storage objects and stream objects. At the same time, structured storage enables end users to interact and manage a compound file as if it were a single file rather than a nested hierarchy of separate objects.

Structured storage also provides additional benefits:

- Incremental access. If a user needs access to an object within a compound file, the user can load and save only that object, rather than the entire file.
- Multiple use. More than one end user or application can concurrently read and write information in the same compound file.
- Transaction processing. Users can read or write to OLE compound files in transacted mode, where changes made to the file are buffered and can subsequently either be committed to the file or reversed.
- Low memory saves. Structured storage provides facilities for saving files in low memory situations.

Compound Files

Although you can implement your own structured storage objects and interfaces, OLE provides a standard implementation called Compound Files. Using Compound Files saves you the work of coding your own implementation of structured storage and confers several additional benefits derived from adhering to a defined standard. These benefits include the following:

- File-system and platform independence. Since OLE's Compound Files implementation runs on top of existing flat file systems, compound files stored in FAT, NTFS, or the Macintosh file systems can be opened by applications using any one of the others.
- Browsability. Since the separate objects in a compound file are saved in a standard format and can be accessed using standard OLE interfaces and APIs, any browser utility using these interfaces and APIs can list the objects in the file, even though the data within a given object may be in a proprietary format.
- Access to certain internal data. Since the Compound Files implementation provides standard ways of writing certain types of data – summary information, for example – applications can read this data using OLE interfaces and APIs.

Structured Storage Elements

Structured storage provides the equivalent of a complete file system for storing objects through the following elements:

- The **IStorage**, **IStream**, **ILockBytes**, and **IRootStorage** interfaces, along with a set of related interfaces − **IPersistStorage**, **IPersist**, **IPersistFile**, and **IPersistStream**.
- Helper APIs that facilitate your implementation of structured storage, as well as a second set of APIs for compound files, OLE's implementation of the structured storage interfaces. OLE also provides a set of supporting structures and enumeration values used to organize parameters for interface methods and APIs.
- A set of access modes for regulating access to compound files.

Interfaces

Structured storage services are organized into three categories of interfaces. Each set represents a successive level of indirection, or abstraction, between a compound file, the objects it contains, and the physical media on which these individual components are stored.

The first category of interfaces consists of **IStorage**, **IStream**, and **IRootStorage**. The first two interfaces define how objects are stored *within* a compound file. These interfaces provide methods for opening storage elements, committing and reverting changes, copying and moving elements, and reading and writing streams. These interfaces do not understand the native data formats of the individual objects and therefore have no methods for saving those objects to persistent storage. The **IRootStorage** interface has a single method for associating a compound document with an underlying file system name. Clients are responsible for implementing these interfaces on behalf of their compound files.

The second category of interfaces consists of the **IPersist** interfaces, which objects implement to manage their persistent data. These interfaces provide methods to read the data formats of individual objects and therefore know how to store them. Objects are responsible for implementing these interfaces because clients do not know the native data formats of their nested objects. These interfaces, however, have no knowledge of specific physical storage media.

A third category consists of a single interface, **ILockBytes**, which provides methods for writing files to specific physical media, such as a hard disk or tape drive. OLE provides an **ILockBytes** interface for the operating system's file system.

API Functions

Some of the API functions for structured storage are helper functions that perform a sequence of calls to other API functions and interface methods. You can use these helper functions as short cuts.

Other API functions provide access to OLE's implementation of structured storage, Compound Files.

Still another set of API functions enable you to convert OLE 1 objects to structured storage. You can use these functions to determine if an object class is from OLE 1 and to convert objects between OLE 1 and current OLE storage formats.

Finally, there are API functions for converting and emulating other objects. These functions provide services for one server application to work with data from another application. The data can be converted to the native format of the server application by reading the data from its original format but writing it in the native format of the object application. Or the data can remain in its original format while the server application both reads and writes the data in its original format.

Access Modes

In a world where multiple processes and users can access an object simultaneously, mechanisms for controlling that access are essential. OLE provides these mechanisms by defining access modes for both storage and stream objects. The access mode specified for a parent storage object is inherited by its children, though you can place additional restrictions on the child storage or stream. A nested storage or stream object can be opened in the same mode or in a more restricted mode than that of its parent, but it cannot be opened in a less restricted mode than that of its parent.

You specify access modes by using the values listed in the **STGM** enumeration. These values serve as flags to be passed as arguments to methods in the **IStorage** interface and associated API functions. Typically, several flags are combined in the parameter *grfMode,* using a Boolean OR operation.

The flags fall into six groups: transaction flags, storage creation flags, temporary creation flag, priority flag, access permission flags, and shared access flags. Only one flag from each group can be specified at a time.

Transaction Flags

An object can be opened in either direct or transacted mode. When an object is opened in direct mode, changes are made immediately and permanently. When an object is opened in transacted mode, changes are buffered so they can be explicitly committed or reverted once editing is complete. Committed changes are saved to the object while reverted changes are discarded. Direct mode is the default access mode.

Transacted mode is not required on a parent storage object in order to use it on a nested element. A transaction for a nested element, however, is nested within the transaction for its parent storage object. Therefore, changes made to a child object cannot be committed until those made to the parent are committed, and both are not committed until the root storage object (the top-level parent) is actually written to disk. In other words, the changes move outward: inner objects publish changes to the transactions of their immediate containers.

Storage Creation Flags

Storage creation flags specify what OLE should do if an existing storage, stream, or lockbytes object has the same name as a new storage or stream object that you are creating. The default is to return an error message and not create the new object. You can use only one of these flags in a given creation call.

Temporary Creation Flag

The temporary creation flag indicates that the underlying file is to be automatically destroyed when the root storage object is released. This capability is most useful for creating temporary files.

Priority Flag

The priority flag opens a storage object in priority mode. When it opens an object, an application usually works from a snapshot copy because other applications may also be using the object at the same time. When opening a storage object in priority mode, however, the application has exclusive rights to commit changes to the object.

Priority mode enables an application to read some streams from storage before opening the object in a mode that would require the system to make a snapshot copy to be made. Since the application has exclusive access, it doesn't have to make a snapshot copy of the object. When the application subsequently opens the object in a mode where a snapshot copy is required, the application can exclude from the snapshot the streams it has already read, thereby reducing the overhead of opening the object.

Since other applications cannot commit changes to an object while it is open in priority mode, applications should keep it in that mode for as short a time as possible.

Access Permission Flags

Access permission flags specify the type of access a client application has to an open object: read, write, or read/write. Read permission enables an application to read the contents of a stream but not to write changes to the stream. Write permission enables an application to call a function that commits changes to an object's storage but not to read the contents of the stream. Read/write permission enables an application to either read the object's streams or write to the object's storage.

Shared Access Flags

The shared access flags specify the degree of access other applications can have to an object that your application is opening. You can deny read access, deny write access, or deny all access. You can also specify explicitly that no type of access be denied.

Storage Object Naming Conventions

Storage and stream objects are named according to a set of conventions.

The name of a root storage object is the actual name of the file in the underlying file system. It obeys the conventions and restrictions the file system imposes. Filename strings passed to storage-related methods and functions are passed on, uninterpreted and unchanged, to the file system.

The name of a nested element contained within a storage object is managed by the implementation of the particular storage object. All implementations of storage objects must support nested element names 32 characters in length (including the NULL terminator), although some implementations might support longer names. Whether the storage object does any case conversion is implementation-defined. As a result, applications that define element names must choose names that are acceptable in either situation. The OLE implementation of compound files supports names up to 32 characters in length, and does not perform any case conversion.

Persistent Property Sets

While the kind of run-time properties that Automation and ActiveX Controls offer are important, they do not directly address the need to store information with objects persistently stored in the file system. These entities could include files (structured, compound, etc.), directories, and summary catalogs. OLE provides both a standard serialized format for these persistent properties, and a set of interfaces and functions that allow you to create and manipulate the property sets and their properties.

Persistent properties are stored as sets, and one or more sets may be associated with a file system entity. These persistent property sets are intended to be used to store data that is suited to being represented as a collection of fine-grained values. They are not intended to be used as a large data base. They can be used to store summary information about an object on the system, which can then be accessed by any other object that understands how to interpret that property set.

Previous versions of OLE specified very little with respect to properties and their usage, but did define a serialized format that allowed developers to store properties and property sets in an **IStorage** instance. The property identifiers and semantics of a single property set, used for summary information about a document, was also defined. At that time, it was necessary to create and manipulate that structure directly as a data stream. For information on the property set serialized data format structure, refer to **OLE Serialized Property Set Format**.

Now, however, OLE defines two primary interfaces to manage property sets:

- **IPropertyStorage**
- **IPropertySetStorage**

It is no longer necessary to deal with the serialized format directly when these interfaces are implemented on an object that supports the **IStorage** interface (such as compound files). Writing properties through **IPropertySetStorage** and **IPropertyStorage** creates data that exactly conforms to the OLE property set format, as viewed through **IStorage** methods. The converse is also true—properties written to the OLE property set format using **IStorage** are visible through **IPropertySetStorage** and **IPropertyStorage** (although you cannot expect to write to IStream and have the properties through **IPropertyStorage** immediately available, or vice versa).

The **IPropertySetStorage** interface defines methods that create and manage property sets. The **IPropertyStorage** interface directly manipulates the properties within a property set. By calling the methods of these interfaces, an application developer can manage whatever property sets are appropriate for a given file system entity. Use of these interfaces provides one tuned reading and writing implementation for properties, rather than having an implementation in each application, which could suffer performance bottlenecks such as incessant seeking. You can implement the interfaces to enhance performance, so properties can be read and written more quickly by, for example, more efficient caching. Furthermore, **IPropertyStorage** and **IPropertySetStorage** make it possible to manipulate properties on entities that do not support **IStorage**, although in general, most applications will not do so.

Managing Property Sets

A persistent property set contains related pieces of information in the form of properties. Each property set is identified with a FMTID, a GUID that allows programs accessing the property set to identify the property set and, through this identification, know how to interpret the properties it contains. Examples of property sets might be the character-formatting properties in a word processor or the rendering attributes of an element in a drawing program.

OLE defines the **IPropertySetStorage** interface to facilitate management of property sets. Through the methods of this interface, you can create a new property set, or open or delete an existing property set. In addition, it provides a method that creates an enumerator and supplies a pointer to its **IEnumSTATPROPSETSTG** interface. You can call the methods of this interface to enumerate **STATPROPSETSTG** structures on your object, which will provide information about all of the property sets on the object.

When you create or open an instance of **IPropertyStorage**, it is similar to opening an object that supports **IStorage** or **IStream**, because you need to specify the storage mode in which you are opening the interface. For **IStorage**, these include the transaction mode, the read/write mode, and the sharing mode.

When you create a property set with a call to **IPropertySetStorage::Create**, you specify whether the property set is to be simple or non-simple. A simple property set contains types that can be fully written within the property set stream, which is intended to be limited, and can, in fact, not be larger than 256 Kbytes. However, for those cases when you need to store a larger amount of information in the property set, you can specify that the property set be non-simple, allowing you to use one or more of the types that specify only a pointer to a storage or stream object. Also, if you need a transacted update, the property set must be non-simple. There is, of course, a certain performance penalty for opening these types, because it requires opening the stream or storage object to which you have the pointer.

If your application uses compound files, you can use the OLE-provided implementation of these interfaces, which are implemented on the OLE compound file storage object.

Each property set consists primarily of a logically connected group of properties, as described in the following section.

Managing Properties

Every property consists of a *property identifier* (unique within its property set), a *type tag* that represents the type of a value, and the *value* itself. The type tag describes the representation of the data in the value. In addition, a property may also be assigned a string name that can be used to identify the property, rather than using the required numerical property identifier. To create and manage properties, OLE defines the **IPropertyStorage** interface.

The **IPropertyStorage** interface includes methods to read and write arrays of either properties themselves or just property names. The interface includes **Commit** and **Revert** methods that are similar to **IStorage** methods of the same name. There are utility methods that allow you to set the CLSID of the property set, the times associated with the set, and get statistics about the property set. Finally, the **Enum** method creates an enumerator and returns a pointer to its **IEnumSTATPROPSTG** interface. You can call the methods of this interface to enumerate **STATPROPSTG** structures on your object, which will provide information about all of the properties in the current property set.

To illustrate how properties are represented, if a specific property in a property set holds an animal's scientific name, that name could be stored as a zero-terminated string. Stored along with the name would be a type indicator to indicate that the value is a zero-terminated string. These properties might have the following characteristics:

Property ID	String Identifier	Type Indicator	Value Represented
02	PID_ANIMALNAME	VT_LPWSTR	Zero-terminated Unicode string
03	PID_LEGCOUNT	VT_I2	WORD

Any application that recognizes the property set format (identifying it through its FMTID) can look at the property with an identifier of PID_ANIMALNAME, determine it is a zero-terminated string, and read and write the value. While the application can call **IPropertyStorage::ReadMultiple** to read any or all of a property set (having first obtained a pointer), the application must know how to interpret the property set.

A property value is passed through property interfaces as an instance of the type **PROPVARIANT**.

It is important to distinguish between these stored (persistent) properties, and run-time properties. Value type constants have names beginning with **VT_**. The set of valid PROPVARIANTs is, however, not completely equivalent with the set of VARIANTs used in Automation and ActiveX Controls.

The only difference between the two structures is the allowable set of VT_ tags in each. Where a certain property type can be used in both a VARIANT and a PROPVARIANT, the type tag (the VT_ value) always has an identical value. Further, for a given VT_ value, the in-memory representation used in both VARIANTs and PROPVARIANTs is identical. Taken all together, this approach allows the type system to catch disallowed type tags, while at the same time, allowing a knowledgeable client simply to do a pointer-cast when appropriate.

Using Property Sets

While the potential for uses of persistent property sets is not fully tapped, there are currently two primary uses:

- Storing summary information with an object such as a document
- Transferring property data between objects

OLE property sets were designed to store data that is suited to representation as a moderately sized collection of fine-grained values. Data sets that are too large for this to be feasible should be broken into separate streams, storages, and/or property sets. The OLE property set data format was not meant to provide a substitute for a database of many tiny objects.

This section discusses two ways to use property sets. The first describes an example of storing property sets within files to allow common access to the information in the property set, and describes the "OLE summary information" property set standard. The second is an example that shows how to transfer property sets between applications or OLE objects as an effective means of communication.

Storing Information with System Objects

One of the most common current uses of persistent properties is to use them to store information about a system object, such as a document, with that object. There is, of course, the potential of storing properties with any object, such as a printer, so it would only be necessary to look at its properties to determine its location, its type, and so on. A user object could have a property set that includes information like First Name, Last Name, Office, and Phone. Applications can be written to query a system-wide set of objects based on their properties, for example, displaying all printers located in a certain building. Current systems, however, most frequently use properties on documents.

The primary property set standard that OLE has defined is the Summary Information property set. This property set is both simple and commonly used. Most documents created by applications have a common set of attributes that are useful to users of those documents. These attributes include the name of the document's author, the subject of the document, when it was created, and so on. Two other property sets are defined for Office95. These are the OLE Document Summary property set and the User-Defined Properties property set, and are described in more detail in Appendix C, **OLE Serialized Property Set Format**.

In Windows 3.1, each application had a different way of storing this information within its documents. To examine the summary information for a given document, the user had to run the application that created the document, open it, and invoke the application's Summary Information dialog box, so only the application could display its summary information.

OLE property sets and the property set interfaces make it possible to see a document's properties without running the creating application. For example, Microsoft Word 6.0 and many other OLE-enabled applications now save their documents using OLE structured storage and the property set standard described here. Thus, applications other than Word 6.0 are able to display the summary information property set for such a file, as long as that file is an OLE structured storage file, and the creating application saved the information in the OLE Property Set format. The Windows 95 shell, for example, takes advantage of this, and allows the end user to see the properties of any Word 6.0 document directly from the shell.

To take advantage of property sets from other applications, the other applications must understand how to interpret the properties within a property set, which implies a standard. OLE has pioneered this approach by defining one standard property set, the OLE Summary Information Property Set. Any application that has the definition of this property set can easily access the summary information contained in any document created by an OLE application that uses that property set specification.

The following section describes the Summary Information property set as an example of property set definition.

The Summary Information Property Set

OLE defines a standard common property set for storing summary information about documents. The Summary Information property set must be stored in an **IStream** instance off of the root storage object; it is not valid to store the property set in the "Contents" stream of a named **IStorage** instance.

For example, to create an ANSI simple property set, you would call **IPropertySetStorage::Create** to create the property set, specifying PROPSETFLAG_ANSI (simple is the default mode), then write to it with a call to **IPropertyStorage::WriteMultiple**. To read the property set, you would call **IPropertyStorage::ReadMultiple**.

All shared property sets are identified by a stream or storage name with the prefix "\005" (or 0x05) to show it is a property set shareable among applications, and the Summary Information property set is no exception. The name of the stream that contains the Summary Information property set is:

```
"\005SummaryInformation"
```

The FMTID for the Summary Information property set is:

```
F29F85E0-4FF9-1068-AB91-08002B27B3D9
```

Use the DEFINE_GUID macro to define the FMTID for the property set:

```
DEFINE_GUID(FormatID_SummaryInformation, 0xF29F85E0, 0x4FF9, 0x1068, 0xAB,
0x91, 0x08, 0x00, 0x2B, 0x27, 0xB3, 0xD9);
```

On an Intel byte-ordered machine, the FMTID has the following representation:

```
E0 85 9F F2 F9 4F 68 10 AB 91 08 00 2B 27 B3 D9
```

The following table shows the string property names for the Summary Information property set, along with the respective property identifiers and VT type indicators.

Property Name	Property ID String	Property ID	VT Type
Title	PID_TITLE	0x00000002	VT_LPSTR
Subject	PID_SUBJECT	0x00000003	VT_LPSTR
Author	PID_AUTHOR	0x00000004	VT_LPSTR
Keywords	PID_KEYWORDS	0x00000005	VT_LPSTR
Comments	PID_COMMENTS	0x00000006	VT_LPSTR
Template	PID_TEMPLATE	0x00000007	VT_LPSTR
Last Saved By	PID_LASTAUTHOR	0x00000008	VT_LPSTR
Revision Number	PID_REVNUMBER	0x00000009	VT_LPSTR
Total Editing Time	PID_EDITTIME	0x0000000A	VT_FILETIME (UTC)
Last Printed	PID_LASTPRINTED	0x0000000B	VT_FILETIME (UTC)
Create Time/Date (*)	PID_CREATE_DTM	0x0000000C	VT_FILETIME (UTC)
Last saved Time/Date (*)	PID_LASTSAVE_DTM	0x0000000D	VT_FILETIME (UTC)
Number of Pages	PID_PAGECOUNT	0x0000000E	VT_I4
Number of Words	PID_WORDCOUNT	0x0000000F	VT_I4
Number of	PID_CHARCOUNT	0x00000010	VT_I4

Characters			
Thumbnail	PID_THUMBNAIL	0x00000011	VT_CF
Name of Creating Application	PID_APPNAME	0x00000012	VT_LPSTR
Security	PID_SECURITY	0x00000013	VT_I4

* Some methods of file transfer (such as a download from a BBS) do not maintain the file system's version of this information correctly.

Guidelines for Implementing the Summary Information Property Set

The following guidelines pertain to implementing the Summary Information property set described in the preceding section:

- PID_TEMPLATE refers to an external document containing formatting and styling information. The means by which the template is located is implementation-defined.
- PID_LASTAUTHOR is the name stored in User Information by the application. For example, suppose Mary creates a document on her machine and gives it to John, who then modifies and saves it. Mary is the author, John is the last saved by value.
- PID_REVNUMBER is the number of times the File/Save command has been called on this document.
- Each of the date/time values must be stored in Universal Coordinated Time (UTC).
- PID_CREATE_DTM is a read-only property; this property should be set when a document is created, but should not be subsequently changed.
- For PID_THUMBNAIL, applications should store data in CF_DIB or CF_METAFILEPICT format. CF_METAFILEPICT is recommended.
- PID_SECURITY is the suggested security level for the document. By noting the security level on the document, an application other than the originator of the document can adjust its user interface to the properties appropriately. An application should not display any information about a password-protected document or allow modifications to enforced read-only or locked-for-annotations documents. Applications should warn the user about read-only recommended if the user attempts to modify properties:

Security Level	Value
None	0
Password protected	1
Read-only recommended	2
Read-only enforced	4
Locked for annotations	8

OLE Compound File Property Set Implementations

OLE provides a compound file implementation of the property set interfaces, along with three helper functions. These implementations are available through OLE compound file storage objects, to which you can get a pointer through the OLE functions **StgCreateDocfile**, to create a new compound file storage object, and **StgOpenStorage**, to open one that currently exists. The following section describes some performance characteristics of these implementations. For more information on specific interfaces and how to get a pointer to these interfaces, refer to the following in the OLE reference section:

- **IPropertySetStorage -- Compound File Implementation**
- **IPropertyStorage -- Compound File Implementation**
- **IEnumSTATPROPSTG -- Compound File Implementation**
- **IEnumSTATPROPSETSTG -- Compound File Implementation**

In addition, there are three helper functions, designed to aid in dealing with propvariants:

- **PropVariantClear**
- **FreePropVariantArray**
- **PropVariantCopy**

Performance Characteristics

A call to the OLE compound file implementation of **IPropertySetStorage** interface to create a property set causes either a stream or storage to be created through a call to **IStorage::CreateStream** or **IStorage::CreateStorage**. A default property set is created in memory, but not flushed to disk. When there is a call to **IPropertyStorage::WriteMultiple**, it operates within the buffer.

When a property set is opened, **IStorage::OpenStream** or **IStorage::OpenStorage** is used. The *entire* property set stream is read into contiguous memory. **IPropertyStorage::ReadMultiple** operations then operate by reading the memory buffer. Therefore, the first access is expensive in terms of time (because of disk reads) but subsequent accesses are very efficient. Writes may be slightly more expensive because SetSize operations on the underlying stream may be required to guarantee that disk space is available if data is added.

No guarantees are made as to whether **IPropertyStorage::WriteMultiple** will flush updates. In general, the client should assume that **IPropertyStorage::WriteMultiple** only updates the in memory buffer. To flush data, **IPropertyStorage::Commit** or **IPropertyStorage::Release** (last release) should be called.

This design means that **WriteMultiple** may succeed but the data is not actually persistently written.

Note This size of the property set stream may not exceed 256K bytes.

Using OLE-Implemented Property Sets

Since the property set stream is read into memory in its entirety before a single property can be read or written, it is strongly recommended that property sets be kept small. "Small" is somewhere under 32K of data. This should not present too much of a problem because typically, "in-line" properties will be small items such as descriptive strings, keywords, timestamps, counts, author names, GUIDs, CLSIDs, etc.

For the storage of larger chunks of data, or where the total size of a set of related properties far exceeds the recommended amount, the use of non-simple types such as **VT_STREAM** and **VT_STORAGE** are strongly recommended. These are not stored inside the property set stream, so do not significantly affect the initial overhead of the first accessing/writing of a property. There is *some* effect because the property set stream contains the name of the sibling stream- or storage-valued property and this takes a small amount of time to process.

IPropertySetStorage Implementation Considerations

Several issues arise when considering how to provide an implementation of the **IPropertySetStorage** interface that reads and writes the OLE property set format. The following sections describe these considerations.

Names in IStorage

This is the most complex interoperability issue. In the **IPropertySetStorage** interface, property sets are identified with FMTIDs, but in **IStorage**, they are named with strings with a maximum length of 32 characters.

To accomplish this mapping, the first task is to establish a mapping between FMTIDs and strings. Converting in the one direction, you have a FMTID, and need a corresponding string name. First, check whether the FMTID is one of a fixed set of well-known values, and use the corresponding well-known string name if so:

FMTID	String Name	Semantic
F29F85E0-4FF9-1068-AB91-08002B27B3D9	"\005SummaryInformation"	OLE2 summary information
D5CDD502-2E9C-101B-9397-08002B2CF9AE D5CDD505-2E9C-101B-9397-08002B2CF9AE	"\005DocumentSummaryInformation"	Office document summary information and user-defined properties.

Note The DocumentSummaryInformation property set is special in that it contains two sections. Multiple sections are not permitted in any other property set. This property set is described in more detail in Appendix B, **OLE Serialized Property Set Format**.

The first was defined as part of OLE; the second one was defined by Microsoft Office.

Otherwise, algorithmically form a string name in the following way. First, convert the FMTID to little-endian byte order if necessary. Then, take the 128 bits of the FMTID and consider them as one long bit string by concatenating each of the bytes together. The first bit of the 128 bit value is the least significant bit of the first byte in memory of the FMTID; the last bit of the 128 bit value is the most significant bit of the last byte in memory of the FMTID. Extend these 128 bits to 130 bits by adding two zero bits to the end. Next, chop the 130 bits into groups of five bits; there will be 26 such groups. Consider each group as an integer with reversed bit precedence. For example, the first of the 128 bits is the least significant bit of the first group of five bits; the fifth of the 128 bits is the most significant bit of the first group. Map each of these integers as an index into the array of thirty-two characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ012345

This yields a sequence of 26 Unicode characters that uses only uppercase characters and numerals. Note that no two characters in this range compare equally in a case-insensitive manner in any locale. The final string is the concatenation of the string "\005" onto the front of these 26 characters, for a total length of 27 characters.

The following code illustrates one way to map from FMTID to property string:

```
#define CBIT_BYTE        8
#define CBIT_CHARMASK    5
#define CCH_MAP          (1 << CBIT_CHARMASK)     // 32
#define CHARMASK         (CCH_MAP - 1)            // 0x1f

CHAR awcMap[CCH_MAP + 1] = "abcdefghijklmnopqrstuvwxyz012345";

WCHAR MapChar(ULONG i) {
    return((WCHAR) awcMap[i & CHARMASK]);
```

```
    }

VOID GuidToPropertyStringName(GUID *pguid, WCHAR awcname[]) {
    BYTE *pb = (BYTE *) pguid;
    BYTE *pbEnd = pb + sizeof(*pguid);
    ULONG cbitRemain = CBIT_BYTE;
    WCHAR *pwc = awcname;

    *pwc++ = ((WCHAR) 0x0005);
    while (pb < pbEnd) {
        ULONG i = *pb >> (CBIT_BYTE - cbitRemain);
        if (cbitRemain >= CBIT_CHARMASK) {
            *pwc = MapChar(i);
            if (cbitRemain == CBIT_BYTE && *pwc >= L'a' && *pwc <= L'z') {
                *pwc += (WCHAR) (L'A' - L'a');
                }
            pwc++;
            cbitRemain -= CBIT_CHARMASK;
            if (cbitRemain == 0) {
                pb++;
                cbitRemain = CBIT_BYTE;
                }
            }
        else {
            if (++pb < pbEnd) {
                i |= *pb << cbitRemain;
                }
            *pwc++ = MapChar(i);
            cbitRemain += CBIT_BYTE - CBIT_CHARMASK;
            }
        }
    *pwc = L'\0';
    }
```

Converters of property string names to GUIDs should accept lowercase letters as synonymous with their upper case counterparts. The following example shows one way to map from the property string to a FMTID:

```
ULONG
PropertySetNameToGuid(
    IN ULONG cwcname,
    IN WCHAR const awcname[],
    OUT GUID *pguid)
{
    ULONG Status = ERROR_INVALID_PARAMETER;
    WCHAR const *pwc = awcname;

    if (pwc[0] == WC_PROPSET0)
    {
        //Note: cwcname includes the WC_PROPSET0, and
        //sizeof(wsz…) includes the trailing L'\0', but
        //the comparison excludes both the leading
        //WC_PROPSET0 and the trailing L'\0'.

        if (cwcname == sizeof(wszSummary)/sizeof(WCHAR) &&
```

```
    wcsnicmp(&pwc[1], wszSummary, cwcname - 1) == 0)
{
    *pguid = guidSummary;
    return(NO_ERROR);
}

if (cwcname == CWC_PROPSET)
{
    ULONG cbit;
    BYTE *pb = (BYTE *) pguid - 1;

    ZeroMemory(pguid, sizeof(*pguid));
    for (cbit = 0; cbit < CBIT_GUID; cbit +=
    CBIT_CHARMASK)
    {
        ULONG cbitUsed = cibt % CBIT_BYTE;
        ULONG cbitStored;
        WCHAR wc;

        if (cbitUsed == 0)
        {
            pb++;
        }
        wc = *++pwc - L'A';        //assume uppercase
        if (wc > CALPHACHARS)
        {
            wc += (WCHAR) (L'A' - L'a')' //try lowercase

            if (wc > CALPHACHARS)
            {
                wc += L'a' - L'0' + CALPHACHARS; //must
                                                 be a digit
                if (wc > CHARMASK)
                {
                    goto fail;       //invalid character
                }
            }
        }
        *pb |= (BYTE) (wc << cbitUsed);
        cbitStored = min(CBIT_BYTE - cbitUsed,
        CBIT_CHARMASK);
        //If the translated bits wouldn't fit in the
          current byte

        if (cbitStored < CBIT_CHARMASK)
        {
            wc >>= CBIT_BYTE - cbitUsed;
            if (cbit + cbitStored == CBIT_GUID)
            {
               if (wc !+0)
               {
                   goto fail;        //extra bits
               }
               break;
            }
```

```
                    pb++;
                    *pb |= (BYTE) wc;
                }
            }
            Status = NO_ERROR
        }
    }
fail:
    return(Status);
}
```

When attempting to open an existing property set (in **IPropertySetStorage::Open**) the (root) FMTID in hand is converted to a string as depicted above. If an element of the **IStorage** of that name exists, it is used. Otherwise, the open fails.

When creating a new property set, the above mapping determines the string name used.

Storage vs Stream for a Property Set

To provide applications the control they need to fully interoperate through the **IPropertySetStorage** interface with the OLE property set, the programmer must control whether a property set is stored in a storage or a stream. This is provided through the presence or absence of the **PROPSETFLAG_NONSIMPLE** flag in **IPropertySetStorage::Create**.

Setting the CLSID of the Property Set

IPropertyStorage::SetClass, when invoked on a property stored in a compound file, will set the CLSID of the storage object through a call to **IStorage::SetClass** in addition to setting the class tag value stored in the OLE property set. This provides a consistency and uniformity that creates better interaction with some tools.

Synchronization Points

When property sets are supported on the same object as is **IStorage**, it is important to be aware of synchronization points between the base storage and the property storage. The property set storage holds the property set stream in an internal buffer until that buffer is commited through the **IPropertyStorage::Commit** method. This is true whether **IPropertyStorage** was opened in transacted mode or direct mode.

Code pages: Unicode strings, Macintosh, etc.

Another consideration is how Unicode property names are stored in the property ID 0 (the property name dictionary), which is not specified per se to use Unicode strings.

This is straightforward. Unicode officially has a code page value of 1200. To store Unicode values in the property name dictionary, use a code page value of 1200 for the whole property set (in property ID 1, of course), specified by the absence of the PROPSETFLAG_ANSI flag in **IPropertySetStorage::Create**. Note that this has the side effect of storing *all* string values in the property set in Unicode. In all code pages, the count found at the start of a VT_LPSTR s a *byte* count, not a character count. This is necessary to provide for smooth interoperability with down-level clients.

The compound file implementation of **IPropertySetStorage** creates all *new* property sets completely either in Unicode (code page 1200) or in the current system ANSI code page. This is controlled by the absence or presence of the PROPSETFLAG_ANSI flag in the *grfFlags* parameter of **IPropertySetStorage::Create**.

It is recommended that property sets be created or opened as Unicode, by not setting the PROPSETFLAG_ANSI flag in the *grfFlags* parameter of **IPropertySetStorage::Create**. It is also recommended that you avoid using VT_LPSTR values, and use VT_LPWSTR values instead. When the property set code page is Unicode, VT_LPSTR string values are converted to Unicode when stored, and back to multibyte string values when retrieved. When the code page of the property set is not Unicode, property names, VT_BSTR strings, and non-simple property values are converted to multibyte strings when stored, and converted back to Unicode when retrieved. If the property set code page is Unicode, VT_LPSTR string values are converted to Unicode when stored, and back to multibyte string values when retrieved.

The setting of the PROPSETFLAG_ANSI flag as reported through a call to **IPropertyStorage::Stat** simply reflects whether the underlying code page is not Unicode or is Unicode. Note, though, property ID 1 can be explicitly read to learn the code page.

Property ID 1 is accessible through **IPropertyStorage::ReadMultiple**. However, it is read-only, and may not be updated with **WriteMultiple**. Further, it may not be deleted with **DeleteMultiple**.

Dictionary

IPropertyStorage::WritePropertyNames is implemented using the property ID 0 dictionary as described above. Property ID 0 is *not* accessible through **IPropertyStorage::ReadMultiple** or **::WriteMultiple**.

Extensions

Property set extensions as defined in the original OLE property set format have been removed and are not supported, except for the User Defined Properties section in the Document Summary Information property set, described in more detail in Appendix C, **OLE Serialized Property Set Format**.

CHAPTER 9

Asynchronous Storage

Asynchronous storage enhances OLE's structured storage specification to support asynchronous downloading of storage objects on high-latency, slow-link networks such as the Internet. Asynchronous storage enables both new and legacy applications that use compound files to efficiently render their content when accessed by means of existing Internet protocols. A single request to a World Wide Web server triggers the download of nested objects contained within a Web page, thereby eliminating the need to separately request each object. An asynchronous download and access mechanism enables an application to render the first page of data before all the data has been received. The exact order in which elements of a page become available can be specified by the Web publisher and is not dependent on random factors of network topology and server availability.

Asynchronous storage works together with asynchronous monikers to provide complete asynchronous binding behavior. (For more information on asynchronous monikers, see the Microsoft's ActiveX™ Development Kit.) A protocol-specific asynchronous moniker triggers the binding operation and sets up the required components. In the Internet case, this moniker would be one that can parse a Universal Resource Locator (URL) to bind to an object or storage. If the target of the binding operation is a persistent object, the call to **IMoniker::BindToStorage** returns an asynchronous storage object.

> **Note** Microsoft's current version of URL monikers does not support asynchronous storage. A future version will do so.

An asynchronous moniker client requests asynchronous binding by implementing a bind-status callback object and registering it with the bind context. The bind-status callback object exposes the **IBindStatusCallback** interface, which enables the client to specify binding preferences and to receive progress and global data-availability notifications during the course of a binding operation. The asynchronous compound file implementation provides a connection point for **IProgressNotify**, which clients can use to receive specific availability notifications on individual streams.

Storage Modes

Asynchronous storage supports two storage modes: blocking and nonblocking, which a client (either a browser or the object itself) can specify by returning BINDF_ASYNCSTORAGE from the moniker's call to **IBindStatusCallback::GetBindInfo**. If a client specifies BINDF_ASYNCSTORAGE, it receives a pointer to a nonblocking asynchronous storage. Otherwise, it receives a pointer to a blocking asynchronous storage. Even if the client does not request an asynchronous binding operation (by not registering **IBindStatusCallback** with the bind context), the moniker still returns a blocking asynchronous storage, enabling progressive loading for legacy applications.

In nonblocking mode, an asynchronous storage returns E_PENDING when data is unavailable. Upon receiving this message, the client waits for notification that additional data is available before trying again to download it.

In blocking mode, instead of returning E_PENDING, the asynchronous storage blocks the call until new data is available, then unblocks the call and returns the new data. The client must be ready to receive the data. While the thread is blocked, data already passed to the client is fully available to the user.

Blocking mode is necessary because clients unaware of asynchronous storage will not recognize E_PENDING and will assume that an unrecoverable error has occurred. Blocking asynchronous storage enables existing clients to do progressive rendering.

Asynchronous Compound Files

Asynchronous Compound Files, the system-provided implementation of asynchronous storage enables the efficient downloading of compound files from the Internet in general and the Web in particular. The basic architecture of Asynchronous Compound Files is shown in the following diagram.

{ewc msdncd, EWGraphic, bsd23512 0 /a "SDK_STOR.WMF"}

The Asynchronous Compound Files implementation can work with new asynchronous moniker types that understand Internet protocols and can bind to an object identified by a Universal Resource Locator (URL). Such a moniker would return an asynchronous **IStream** or **IStorage** pointer from the client's call to **IMoniker::BindToStorage**.

Compound Files in general are implemented on top of a byte array object, an abstraction of a file that represents an object's data as a flat byte array. The byte array object exposes its functionality through the **ILockBytes** interface. If a byte array supports nonblocking asynchronous storage, it returns E_PENDING to the compound-file implementation, which in turn propagates the error back to the caller.

To keep track of the data available during a download, a byte array that supports asynchronous storage exposes the **IFillLockBytes** interface on a wrapper object provided by the system specifically for this purpose. The downloading code provided by an asynchronous moniker calls this interface to fill the byte array asynchronously, as data is available. The wrapper object also exposes an **ILockBytes** interface, which the Asynchronous Compound Files implementation uses to read and write data from and to the array.

Asynchronous storage and stream objects provide a connection point for the **IProgressNotify** interface, which is implemented by the asynchronous moniker's downloading code. The Asynchronous Compound Files implementation calls **IProgressNotify** to provide the downloader with information about the status of the downloading operation.

How Asynchronous Binding and Storage Work

When a user clicks a link representing a document embedded in a Web page, the following steps occur:

1. The browser calls the **MkParseDisplayName** function, passing the link's URL.
2. **MkParseDisplayName** parses the URL, creates a corresponding aynchronous moniker, and returns a pointer to the moniker's **IMoniker** interface.
3. The browser calls **IsAsyncMoniker** to determine if the moniker is asynchronous, creates a bind context, registers the **IBindStatusCallback** interface with the bind context (only if the moniker is asynchronous), and calls **IMoniker::BindToObject**, passing the bind context.
4. The moniker binds to the object and queries it for the **IPersistMoniker** interface, which indicates whether the object supports asynchronous binding and storage. If the object returns a pointer to **IPersistMoniker**:
   A. The URL moniker calls **IPersistMoniker::Load**, passing its own **IMoniker** pointer to the object.
   B. The object modifies the bind context, chooses whether it wants a blocking or non-blocking storage, registers its own **IBindStatusCallback** and calls **IMoniker::BindToStorage** on the pointer it received through **IPersistMoniker::Load**.
   C. The moniker creates an asynchronous storage, keeps a reference to the wrapper object's **IFillLockBytes** interface, registers the **IProgressNotify** interface on the root storage, and calls **IPersistStorage::Load**, passing the asynchronous storage's **IStorage** pointer. As data arrives (on a background thread) the moniker calls **IFillLockBytes** to fill the **ILockBytes** on the temp file.
   D. The object reads data from the storage and returns from **IPersistMoniker::Load** when it has received sufficient data to consider itself initialized. If the object attempts to read data that has not yet been downloaded, the downloader receives a notification on **IProgressNotify**. Inside the **IProgressNotify::OnProgress** method, the downloading thread either blocks in a modal message loop, or causes the asynchronous storage to return E_PENDING, depending on whether the object has requested a blocking or nonblocking storage.
5. If the object does not implement **IPersistMoniker**, the moniker queries for **IPersistStorage**, which indicates that the object's persistent state is stored in a storage object. If the object returns a pointer to **IPersistStorage**:
   A. The Moniker calls **IMoniker::BindToStorage** on itself, requesting a blocking **IStorage** (because the object is not asynchronous-aware), creates an asynchronous storage, keeps a reference to the wrapper object's **IFillLockBytes** interface, registers the **IProgressNotify** interface on the root storage, and calls **IPersistStorage::Load**, passing the asynchronous storage's **IStorage** pointer. As data arrives (on a background thread) the moniker calls **IFillLockBytes** to fill the **ILockBytes** on the temp file.
   B. The object reads data from storage and returns from **IPersistStorage::Load** when it has received sufficient data to consider itself initialized. If the object attempts to read data that has not yet been downloaded, it receives a notification on **IProgressNotify**. Inside the **IProgressNotify::OnProgress** method, the downloading thread always blocks in a modal message loop.
6. Regardless whether the download is synchronous or asynchronous, the moniker returns from **IMoniker::BindToObject**, and the browser receives the initialized object it asked for.
7. The browser queries for **IOleObject** and hosts the object as a Document Object. (At this point the object may not be initialized completely, but only enough to display something useful, in which case downloading continues in the background.)

Compound File Optimization

Asynchronous storage enables applications to precisely lay out their compound files so that data is available in the order in which applications will need it. If an application requires only part of its data to display a first page of information, this data can be placed at the beginning of the file, even if it logically resides at the end of a stream. Data from different streams can be interleaved. Audio and video data, for example, can be interleaved so that subsequent read operations retrieve both simultaneously, a requirement for multimedia applications.

New applications can optimize their compound files programmatically by calling the **StgOpenLayoutDocfile** API function. This function calls the **ILayoutStorage** interface, which is implemented by the root storage of the new compound files implementation. **ILayoutStorage** has member functions for scripting the layout of data, monitoring downloads to determine the order in which data is accessed, and rewriting the compound file to match the layout specified by scripting or determined by monitoring. The call to rewrite a file must occur before releasing the last pointer to the root storage of the file. Otherwise, the file will not be altered.

Legacy applications can optimize their compound files by using the Docfile Layout Tool (*dflayout.exe*) included in the Win32 SDK. This capability, combined with a blocking storage enable legacy applications to progressively download and render their data.

Data Transfer

The Component Object Model (COM) provides a standard mechanism for transferring data between applications. This mechanism is the *data object*, which is simply any COM object that implements the **IDataObject** interface. Some data objects, such as a piece of text copied to the clipboard, have **IDataObject** as their sole interface. Others, such as compound document objects, expose several interfaces, of which **IDataObject** is simply one. Data objects are fundamental to the working of compound documents, although they also have widespread application outside that OLE technology.

By exchanging pointers to a data object, providers and consumers of data can manage data transfers in a uniform manner, regardless of the format of the data, the type of medium used to transfer the data, or the target device on which it is to be rendered. You can include support in your application for basic clipboard transfers, drag and drop transfers, and OLE compound document transfers with a single implementation of **IDataObject**. Having done that, the amount of code required to accommodate the special semantics of each protocol is minimal.

Data Transfer Interfaces

The **IDataObject** interface provides consumers of data with methods for getting and setting an object's data, determining which formats the object supports, and registering for and receiving notifications when data in the object changes. When obtaining data, a caller can specify the format in which it wants to render the data. The source of the data, however, determines the storage medium, which it returns in an out parameter provided by the caller.

By itself, **IDataObject** supplies all the tools you need to implement Microsoft® Windows® clipboard transfers or compound document transfers in your applications. If you also want to support drag and drop transfers, you need to implement the **IDropSource** and **IDropTarget** **interfaces**along with **IDataObject**.

The **IDataObject** interface combined with OLE clipboard APIs provide all the capabilities of the Microsoft® Win32® clipboard APIs. Using both the platform's clipboard APIs and OLE's is usually redundant and unnecessary. Suppliers of data that support either drag and drop transfers or OLE compound documents *must* implement the **IDataObject** interface. If your application supports only clipboard transfers now, but you intend to add drag and drop or compound documents in later releases, you may want to implement **IDataObject** and the OLE clipboard APIs now in order to minimize the amount of time spent recoding and debugging later. You may also want to implement **IDataObject** in order to utilize transfer media other than global memory.

The following table summarizes which ones to use, depending on what types of data transfer you want to support:

If You Want to Support	You Must Use
Compound documents	**IDataObject**
Drag and drop transfers	**IDataObject**, **IDropSource**, **IDropTarget**, **DoDragDrop** (or the equivalent)
Windows clipboard transfers using global memory exclusively	Windows clipboard APIs
Windows clipboard transfers using exchange mediums other than global memory.	**IDataObject**
Clipboard transfers now but drag and drop or compound documents later	**IDataObject** and the interfaces and function listed above for "Drag and drop transfers"

When a user initiates a data transfer operation, the source application creates an instance of **IDataObject** and through it makes the data available in one or more formats. In a clipboard transfer, the application calls the **OleSetClipboard** function to pass a data-object pointer to OLE. (**OleSetClipboard** also offers standard clipboard data formats for both OLE version 1 and non-OLE applications.) In a drag and drop transfer, the application calls the **DoDragDrop** function instead.

On the receiving side of the transfer, the destination receives the **IDataObject** pointer either as an argument to an invocation of **IDropTarget::Drop** or by calling the **OleGetClipboard** function, depending on whether the transfer is by means of drag and drop or the clipboard. Having obtained this pointer, the destination calls **IDataObject::EnumFormatEtc** to learn what formats are available for retrieval and on what types of media they can be obtained. Armed with this information, the receiving application requests the data with a call to **IDataObject::GetData**.

Data Formats and Transfer Media

Most platforms, including Windows, define a standard protocol for transferring data between applications, based on a set of functions called the clipboard. Applications using these functions can share data even if their native data formats are wildly different. Generally, these clipboards have two significant shortcomings that COM has overcome.

First, data descriptions use only a format identifier, such as the single 16-bit clipboard format identifier on Windows, which means the clipboard can only describe the structure of its data, that is, the ordering of the bits. It can report, "I have a bitmap" or "I have some text," but it cannot specify the target devices for which the data is composed, which views or aspects of itself the data can provide, or which storage media are best suited for its transfer. For example, it cannot report, "I have a string of text that is stored in global memory and formatted for presentation either on screen or on a printer" or "I have a thumbnail sketch bitmap rendered for a 100 dpi dot-matrix printer and stored as a disk file."

Second, all data transfers using the clipboard generally occur through global memory. Using global memory is reasonably efficient for small amounts of data but horribly inefficient for large amounts, such as a 20 MB multimedia object. Global memory is slow for a large data object, whose size requires considerable swapping to virtual memory on disk. In cases where the data being exchanged is going to reside mostly on disk anyway, forcing it through this virtual-memory bottleneck is highly inefficient. A better way would skip global memory entirely and simply transfer the data directly to disk.

To alleviate these problems, COM introduces two new data structures: FORMATETC and STGMEDIUM.

The FORMATETC Structure

The FORMATETC structure is a generalized clipboard format, enhanced to encompass a target device, an *aspect* or view of the data, and a storage medium. A data consumer, such as an OLE container application, passes the FORMATETC structure as an argument in calls to **IDataObject** to indicate the type of data it wants from a data source, such as a compound document object. The source uses the FORMATETC structure to describe what formats it can provide. FORMATETC can describe virtually any data, including other objects such as monikers. A container can ask one of its embedded objects to list its data formats by calling **IDataObject::EnumFormatetc**, which returns an enumerator object that implements the **IEnumFormatEtc** interface. Instead of replying merely that it has "text and a bitmap," the object can provide a detailed description of the data, including the device (normally screen or printer) for which it is rendered, the aspect to be presented to the user (full contents, thumbnail, icon, or formatted for printing), and the storage medium containing the data (global memory, disk file, storage object, or stream). This ability to tightly describe data will, in time, result in higher quality printer and screen output as well as more efficiency in data browsing, where a thumbnail sketch is much faster to retrieve and display than a fully detailed rendering.

The following table lists fields of the FORMATETC data structure and the information they specify:

Field	Specifies
cfFormat	The format in which the data is to be rendered, which can be a standard clipboard format, a proprietary format, or an OLE format. For more information on OLE formats, see Chapter 5, "Compound Documents."
ptd	A DVTARGETDEVICE structure, which contains enough information about a Windows target device, such as a screen or printer, so that a handle to its device context (hDC) can be created using the Windows **CreateDC** function.
dwAspect	The aspect or view of the data to be rendered; can be the full contents, a thumbnail sketch, an icon, or formatted for printing.
lindex	The part of the aspect that is of interest; for the present, the value must be -1, indicating that the entire view is of interest.
tymed	The data's storage medium, which can be global memory, disk file, or an instance of one of COM's structured-storage interfaces.

The STGMEDIUM Structure

Just as the FORMATETC structure is an enhancement of the Windows clipboard format identifier, so the STGMEDIUM structure is an improvement of the global memory handle used to transfer the data. The STGMEDIUM structure includes a flag, *tymed,* which indicates the medium to be used, and a union comprising pointers and a handle for getting whichever medium is specified in *tymed.*

The STGMEDIUM structure enables both data sources and consumers to choose the most efficient exchange medium on a per-rendering basis. If the data is so big that it should be kept on disk, the data source can indicate a disk-based medium in its preferred format, only using global memory as a backup if that's the only medium the consumer understands. Being able to use the best medium for exchanges as the default improves overall performance of data exchange between applications. For example, if some of the data to be transferred is already on disk, the source application can move or copy it to a new destination, either in the same application or in some other, without having first to load the data into global memory. At the receiving end, the consumer of the data does not have to write it back to disk.

Drag and Drop

"Drag and drop" refers to data transfers in which a mouse or other pointing device is used to specify both the data source and its destination. In a typical drag and drop operation, a user selects the object to be transferred by moving the mouse pointer to it and holding down either the left button or some other button designated for this purpose. While continuing to hold down the button, the user initiates the transfer by dragging the object to its destination, which can be any OLE container. Drag and drop provides exactly the same functionality as the OLE Clipboard copy and paste but adds visual feedback and eliminates the need for menus. In fact, if an application supports Clipboard copy and paste, little extra is needed to support drag and drop.

During an OLE drag and drop operation, the following three separate pieces of code are used:

Drag-and-drop code source	Implementation and use
IDropSource interface	Implemented by the object containing the dragged data, referred to as the *drag source*.
IDropTarget interface	Implemented by the object that is intended to accept the drop, referred to as the *drop target*.
DoDragDrop function	Implemented by OLE and used to initiate a drag and drop operation. Once the operation is in progress, it facilitates communication between the drag source and the drop target.

The **IDropSource** and **IDropTarget** interfaces can be implemented in either a container or in an object application. The role of drag source or drop target is not limited to any one type of OLE application.

The OLE function **DoDragDrop** implements a loop that tracks mouse and keyboard movement until such time as the drag is canceled or a drop occurs. **DoDragDrop** is the key function in the drag and drop process, facilitating communication between the drag source and drop target.

During a drag and drop operation, three types of feedback can be displayed to the user:

Type of feedback	Description
Source feedback	Provided by the drag source, the source feedback indicates the data is being dragged and does not change during the course of the drag. Typically, the data is highlighted to signal it has been selected.
Pointer feedback	Provided by the drag source, the pointer feedback indicates what happens if the mouse is released at any given moment. Pointer feedback changes continually as the user moves the mouse and/or presses a modifier key. For example, if the pointer is moved into a window that cannot accept a drop, the pointer changes to the "not allowed" symbol.
Target feedback	Provided by the drop target, target feedback indicates where the drop is to occur.

Although the Windows 3.1 File Manager does not use OLE to implement drag and drop transfers, it is

nevertheless an example of an application that acts as both a drag source and drop target and that provides all three types of feedback. When a user selects a file to copy, File Manager supplies source feedback by indicating the selection. As the selection is dragged and modifier keys are pressed, the mouse pointer changes. For example, if the user presses the CTRL key, a plus sign is added, indicating that the drop would result in a copy rather than moving the original. When the file is dragged over an area that is not a drop target, the pointer changes appropriately. Target feedback, a line drawn around a file or directory, is provided when the pointer is over an area that is a drop target.

Drag Source Responsibilities

The drag source is responsible for the following tasks:

- Providing a data-transfer object for the drop target that exposes the **IDataObject** and **IDropSource** interfaces.
- Generating pointer and source feedback.
- Determining when the drag operation has been canceled or a drop operation has occurred.
- Performing any action on the original data caused by the drop operation, such as deleting the data or creating a link to it.

The main task is creating a data-transfer object that exposes the **IDataObject** and **IDropSource** interfaces. The drag source might or might not include a copy of the selected data. Including it is not mandatory, but doing so safeguards against inadvertent changes and allows the Clipboard operations code to be identical to the drag and drop code.

While a drag operation is in progress, the drag source is responsible for setting the mouse pointer and, if appropriate, for providing additional source feedback to the user. The drag source cannot provide any feedback that tracks the mouse position other than by actually setting the real pointer (see the Windows **SetCursor** function). This rule must be enforced to avoid conflicts with the feedback provided by the drop target. (A drag source can also be a drop target. When dropping on itself, the source/target can, of course, provide target feedback to track the mouse position. In this case, however, it is the drop target tracking the mouse, not the source.) Based on the feedback offered by the drop target, the source sets an appropriate pointer.

Data Notification

Objects that consume data from an external source sometimes need to be informed when data in that source changes. For example, a stock ticker tape viewer that relies on data in some spreadsheet needs to be notified when that data changes so it can update its display. Similarly, a compound document needs information about data changes in its embedded objects so that it can update its data caches. In cases such as this, where dynamic updating of data is desirable, sources of data require some mechanism of notifying data consumers of changes as they occur without obligating the consumers to drop everything in order to update their data. Ideally, having been notified that a change has occurred in the data source, a consuming object can ask for an updated copy at its leisure.

COM's mechanism for handling asynchronous notifications of this type is an object called an advise sink, which is simply any COM object that implements an interface called **IAdviseSink**. Consumers of data implement the **IAdviseSink**. They register to receive notifications by handing a pointer to the data object of interest.

The **IAdviseSink** interfaces exposes the following methods for receiving asynchronous notifications:

Method	Notifies the Advise Sink that
OnDataChange	Calling object's data has changed.
OnViewChange	Instructions for drawing the calling object have changed.
OnRename	Calling object's moniker has changed.
OnSave	Calling object has been saved to persistent storage.
OnClose	Calling object has been closed.

As the table indicates, the **IAdviseSink** interface exposes methods for notifying the advise sink of events other than changes in the calling object's data. The calling object can also notify the sink when the way in which it draws itself changes, or it is renamed, saved, or closed. These other notifications are used mainly or entirely in the context of compound documents, although the notification mechanism is identical. For more information on compound-document notifications, see "Compound Documents."

In order to take advantage of the advise sink, a data source must implement **IDataObject::DAdvise**, **IDataObject::DUnadvise**, and **IDataObject::EnumDAdvise**. A data consumer calls the **DAdvise** mothod to notify a data object that it wishes to be notified when the object's data changes. The consuming object calls the **DUnadvise** method to tear down this connection. Any interested party can call the **EnumAdvise** method to learn the number of objects having an advisory connection with a data object.

When data changes at the source, the data object calls **IAdviseSink::OnDataChange** on all data consumers that have registered to receive notifications. To keep track of advisory connections and manage the dispatch of notifications, data sources rely on an object called a *data advise holder*. You can create your own data advise holder by implementing the **IDataAdviseHolder** interface. Or, you can let COM do it for you by calling the helper function **CreateDataAdviseHolder**.

Property Pages and Property Sheets

OLE property pages enable an object to display its properties in a tabbed dialog box known as a property sheet. An end user can then view and change the object's properties. An object can display its property pages independent of its client, or the client can manage the display of property pages from a number of contained objects in a single property sheet. Property pages also provide a means for notifying a client of changes in an object's properties.

Any object that wishes to provide a user interface for its changing properties can use this technology.

Property Sheets and Property Pages

An object's properties are exposed to clients the same as methods through either COM interfaces or the object's **IDispatch** implementation, allowing properties to be changed by programs calling these methods. The OLE technology of property pages provides the means to build a user interface for an object's properties according to Windows user interface standards. Thus, the properties are exposed to end users. An object's property sheet is a tabbed-dialog where each tab corresponds to a specific property page. The OLE model for working with property pages consists of these features:

- Each property page is managed by an in-process object that implements either **IPropertyPage** or **IPropertyPage2**. Each page is identified with its own unique CLSID.
- An object specifies its support for property pages by implementing **ISpecifyPropertyPages**. Through this interface the caller can obtain a list of CLSIDs identifying the specific property pages that the object supports. If the object specifies a property page CLSID, the object must be able to receive property changes from the property page.
- Any piece of code (client or object) that wishes to display an object's property sheet passes the object's **IUnknown** pointer (or an array if multiple objects are to be affected) along with an array of page CLSIDs to **OleCreatePropertyFrame** or **OleCreatePropertyFrameIndirect**, which creates the tabbed-dialog box.
- The property frame dialog instantiates a single instance of each property page, using **CoCreateInstance** on each CLSID. The property frame obtains at least an **IPropertyPage** pointer for each page. In addition, the frame creates a property page site object in itself for each page. Each site implements **IPropertyPageSite** and this pointer is passed to each page. The page then communicates with the site through this interface pointer.
- Each page is also made aware of the object or objects for which it has been invoked; that is, the property frame passes the **IUnknown** pointers of the objects to each page. When instructed to apply changes to the objects, each page queries for the appropriate interface pointer and passes new property values to the objects in whatever way is desired. There are no stipulations on how such communication has to happen.
- An object can also support per property browsing through the **IPerPropertyBrowsing** interface permitting the object to specify which property should receive initial focus when the property page is displayed and to specify string's and values that can be displayed by the client in its own user interface.

These features are illustrated in the following diagram:

{ewc msdncd, EWGraphic, bsd23520 0 /a "SDK.WMF"}

These interfaces are defined as follows:

```
interface ISpecifyPropertyPages : IUnknown
    {
    HRESULT GetPages([out] CAUUID *pPages);
    };


interface IPropertyPage : IUnknown
    {
    HRESULT SetPageSite([in] IPropertyPageSite *pPageSite);
    HRESULT Activate([in] HWND hWndParent, [in] LPCRECT prc
        , [in] BOOL bModal);
    HRESULT Deactivate(void);
    HRESULT GetPageInfo([out] PROPPAGEINFO *pPageInfo);
    HRESULT SetObjects([in] ULONG cObjects
```

```
        , [in, max_is(cObjects)] IUnknown **ppunk);
    HRESULT Show([in] UINT nCmdShow);
    HRESULT Move([in] LPCRECT prc);
    HRESULT IsPageDirty(void);
    HRESULT Apply(void);
    HRESULT Help([in] LPCOLESTR pszHelpDir);
    HRESULT TranslateAccelerator([in] LPMSG pMsg);
    }

interface IPropertyPageSite : IUnknown
    {
    HRESULT OnStatusChange([in] DWORD dwFlags);
    HRESULT GetLocaleID([out] LCID *pLocaleID);
    HRESULT GetPageContainer([out] IUnknown **ppUnk);
    HRESULT TranslateAccelerator([in] LPMSG pMsg);
    }
```

The **ISpecifyPropertyPages::GetPages** method returns a counted array of UUID (GUID) values each of which describe the CLSID of a property page that the object would like displayed. Whoever invokes the property sheet with **OleCreatePropertyFrame** or **OleCreatePropertyFrameIndirect** passes this array to the API function. Note that if the caller wishes to display property pages for multiple objects, it must only pass the intersection of the CLSID lists of all the objects to these API functions. In other words, the caller must only invoke property pages that are common to all objects.

In addition, the caller passes the **IUnknown** pointers to the affected objects to the API functions as well. Both API functions create a property frame dialog and instantiate a page site with **IPropertyPageSite** for each page it will load. Through this interface a property page can:

- retrieve the current language used in the property sheet through **GetLocaleID**,
- ask the frame to process keystrokes through **TranslateAccelerator**,
- notify the frame of changes in the page through **OnStatusChange**,
- obtain an interface pointer for the frame itself through **GetPageContainer**, although there are no interfaces defined for the frame at this time for this function always returns E_NOTIMPL

The property frame instantiates each property page object and obtain each page's **IPropertyPage** interface. Through this interface the frame informs the page of its page site (**SetPageSite**), retrieves page dimensions and strings (**GetPageInfo**), passes the interface pointers to the affected objects (**SetObjects**), tells the page when to create and destroy its controls (**Activate** and **Deactivate**), instructs the page to show or reposition itself (**Show** and **Move)**, instructs the page to apply its current values to the affected objects (**Apply**), checks on the page's dirty status (**IsPageDirty**), invokes help (**Help**), and passes keystrokes to the page (**TranslateAccelerator**).

An object can also support per-property browsing which provides:

a) a way (through **IPerPropertyBrowsing** and **IPropertyPage2**) to specify which property on which property page should be given the initial focus when a property sheet is first displayed

b) a way (through **IPerPropertyBrowsing**) for the object to specify predefined values and corresponding descriptive strings that could be displayed in a client's own user interface for properties.

An object can choose to support (b) without supporting (a), such as when the object has no property sheet.

The **IPropertyPage2** and **IPerPropertyBrowsing** interfaces are defined as follows:

```
interface IPerPropertyBrowsing : IUnknown
    {
```

```
    HRESULT GetDisplayString([in] DISPID dispID
        , [out] BSTR *pbstr);
    HRESULT MapPropertyToPage([in] DISPID dispID
        , [out] CLSID *pclsid);
    HRESULT GetPredefinedStrings([in] DISPID dispID
        , [out] CALPOLESTR *pcaStringsOut
        , [out] CADWORD *pcaCookiesOut);
    HRESULT GetPredefinedValue([in] DISPID dispID
        , [in] DWORD dwCookie, [out] VARIANT *pvarOut);
    }

interface IPropertyPage2 : IPropertyPage
    {
    HRESULT EditProperty([in] DISPID dispID);
    }
```

To specify its support for such capabilities, the object implements **IPerPropertyBrowsing**. Through this interface, the caller can request the information necessary to achieve the browsing, such as predefined strings (**GetPredefinedStrings**) and values (**GetPredefinedValues**) as well as a display string for a given property (**GetDisplayString**).

In addition, the client can obtain the CLSID of the property page that allows the user to edit a given property identified with a DISPID (**MapPropertyToPage**). The client then instructs the property frame to activate that page initially by passing the CLSID and the DISPID to **OleCreatePropertyFrameIndirect**. The frame activates that page first and passes the DISPID to the page through **IPropertyPage2::EditProperty**. The page then sets the focus to that property's editing field. In this way, a client can jump from a property name in its own user interface to the property page that can manipulate that property.

Data Binding through IPropertyNotifySink

Objects that support properties, for example, through OLE Automation and the **IDispatch** interface, may wish to allow clients to be notified when certain properties change value. Such a property is called a bindable property because the notifications allow a client to synchronize its own display of the object's current property values. In addition, the same objects may wish to allow a client to control when certain properties are allowed to change. Such properties are called request edit properties.

The **IPropertyNotifySink** is a standard notification interface that supports bindable and request-edit properties. **IPropertyNotifySink** is supported from an object with properties as an outgoing interface. That is, the interface itself is implemented by a client's sink object, and the client connects the sink to the supporting object through the connection point mechanism described earlier. The **IPropertyNotifySink** is defined as follows:

```
interface IPropertyNotifySink : IUnknown
    {
    HRESULT OnChanged([in] DISPID dispID);
    HRESULT OnRequestEdit([in] DISPID dispID);
    }
```

When an object wishes to notify its connected sinks that a bindable property identified with a given DISPID has changed, it calls **OnChanged.** If an object changes multiple properties at once, it can pass DISPID_UNKNOWN to **OnChanged** in which case a client refreshes its cache of all property values of interest.

When a request edit property is about to change, an object can ask the client whether it will allow that change to occur. The object calls **OnRequestEdit** passing the DISPID of the property in question (or DISPID_UNKNOWN to identify all properties). The client's sink returns S_OK to indicate that the change is allowed, or S_FALSE (or an error) to indicate that change is not allowed. When an object calls **OnRequestEdit,** it is required to obey the client's wishes by following the exact semantics of S_OK and S_FALSE return values.

Note that **OnRequestEdit** cannot be used for data validation because at the time of the call, the new value of the property is not yet available. The notification can only be used to control a read-only state for a property.

Objects control which properties are bindable and request edit and mark such properties in the object's type information. In the type information, the attribute **bindable** marks a property as supporting **IPropertyNotifySink::OnChanged**. The attribute **requestedit** marks a property as supporting **IPropertyNotifySink::OnRequestEdit**.

One property can support both behaviors in which case **OnRequestEdit** is called first, and only if change is allowed is **OnChanged** called.

The one exception to the behavior of such properties is that no notifications are sent as a result of an object's initialization or loading procedures. At such times, it is assumed that all properties change and that all must be allowed to change. Notifications to this interface are therefore only meaningful in the context of a fully initialized/loaded object.

Two other attributes can be applied to properties in an object's type information. The **defaultbind** attribute marks a **bindable** property as being the one that best represents the state of the object as a whole. The **displaybind** attribute marks a **bindable** property as suitable for display in a client's own user interface.

Compound Documents

OLE compound documents enable users working within a single application to manipulate data written in various formats and derived from multiple sources. For example, a user might insert into a word processing document a graph created in a second application and a sound object created in a third application. Activating the graph causes the second application to load its user interface, or at least that part containing tools necessary to edit the object. Activating the sound object causes the third application to play it. In both cases, a user is able to manipulate data from external sources from within the context of a single document.

OLE compound document technology rests on a foundation consisting of COM, structured storage, and .uniform data transfer. As summarized below, each of these core technologies plays a critical role in OLE compound documents:

COM

A compound document object is essentially a COM object that can be embedded in, or linked to, an existing document. As a COM object, a compound document object exposes the **IUnknown** interface, through which clients can obtain pointers to its other interfaces, including several, such as **IOleObject**, **IOleLink**, and **IViewObject2,** that provide special features unique to compound document objects.

Structured Storage

A compound document object must implement the **IPersistStorage** or, optionally, **IPersistStream** interfaces to manage its own storage. A container used to create compound documents must supply the **IStorage** interface, through which objects store and retrieve data. Containers almost always provide instances of IStorage obtained from OLE's Compound Files implementation. Containers must also use an object's IPersistStorage and/or IPersistStream interfaces.

Uniform Data Transfer

Applications that support compound documents must implement **IDataObject** because embedded objects and linked objects begin as data that has been transferred using special OLE clipboard formats, rather than standard Microsoft® Windows® clipboard formats. In other words, formatting data as an embedded or linked object is simply one more option provided by OLE's uniform data transfer model.

OLE's compound document technology benefits both software developers and users alike. Instead of feeling obligated to cram every conceivable feature into a single application, software developers are now free, if they like, to develop smaller, more focused applications that rely on other applications to supply additional features. In cases where a software developer decides to provide an application with capabilities beyond its core features, the developer can implement these additional services as separate DLLs, which are loaded into memory only when their services are required. Users benefit from smaller, faster, more capable software that they can mix and match as needed, manipulating all required components from within a single master document.

Containers and Servers

Compound document applications are of two basic types: container applications and server applications. OLE container applications provide users with the ability to create, edit, save, and retrieve compound documents. OLE server applications provide users with the means to create documents and other data representations that can be contained as either links or embeddings in container applications. An OLE application can be a container application, a server application, or both.

OLE server applications also differ in whether they are implemented as *in-process servers* or *local servers*. An in-process server is a dynamic link library (DLL) that runs in the container application's process space. You can run an in-process server *only* from within the container application.

Note Future releases of OLE will enable linking and embedding across machine boundaries, so that a container application on one computer will be able to use a compound document object provided by a *remote server* running on another computer. From a container application's point of view, any OLE server application that runs in its own process space, whether on the same computer or a remote machine, is an *out-of-process server.*

Linking and Embedding

Users can create two types of compound-document objects: *linked* or *embedded*. The difference between the two types lies in how and where the object's source data is stored. Where the object resides affects, in turn, the object's portability and methods of activation, how data updates are performed, and the size and structure of its container.

Linked Objects

When a link to an object is inserted in a compound document, the source data, or *link source*, continues to reside wherever it was initially created, usually in another document. The compound document contains only a reference, or *link*, to the actual data stored at the link source, along with information about how to present that data to the user. Currently, moving a link source breaks the link unless both source and client maintain their relative positions in the directory tree. Eventually, link tracking by Windows will allow a link source to be moved independently of its client without breaking the link.

Activating a link runs the link source's server application, which the user requires in order to edit or otherwise manipulate the link data. Linking keeps the size of a compound document small. It is also useful when the data source is maintained by someone else and must be shared among many users. If the person maintaining the link source changes the data, the change is automatically updated in all documents containing a link to that data. In addition to creating simple links, users can nest links and combine linked and embedded objects to create complex documents.

Embedded Objects

An embedded object is physically stored in the compound document, along with all the information needed to manage the object. In other words, the embedded object is actually a part of the compound document in which it resides. This arrangement has a couple of disadvantages. First, a compound document containing embedded objects will be larger than one containing the same objects as links. Second, changes made to the source of an embedded object will not be automatically replicated in the embedded copy, and changes in the source will not be reflected in the source, as they are with a link.

Still, for certain purposes, embedding offers several advantages over links. First, users can transfer compound documents with embedded objects to other computers, or other locations on the same computer, without breaking a link. Second, users can edit embedded objects without changing the content of the original. Sometimes, this separation is precisely what is required. Third, embedded objects can be activated in place, meaning that the user can edit or otherwise manipulate the object without having to work in a separate window from that of the object's container. Instead, when the object is activated, the container application's user interface changes to expose those tools that are necessary to manage or modify the object.

Object Handlers

If an OLE server application is a local server, meaning that it runs in its own process space, communication between container and server must occur across process boundaries. Since this process is expensive, OLE relies on a surrogate object loaded into the container's process space to act on behalf of a local server application. This surrogate object, known as an *object handler* services container requests that do not require the attention of the server application, such as requests for drawing. When a container requests something that the object handler cannot provide, the handler communicates with the server application using COM's out-of-process communication mechanism.

An object handler is unique to an object class. When you create an instance of a handler for one class, you cannot use it for another. When used for a compound document, the object handler implements the container-side data structures when objects of a particular class are accessed remotely.

OLE provides a default object handler that local server applications can use. For applications that require special behaviors, developers can implement a custom handler that either replaces the default handler or uses it to provide certain default behaviors.

An object handler is a DLL containing several interacting components. These components include remoting pieces to manage communication between the handler and its server application, a cache for storing an object's data, along with information on how that data should be formatted and displayed, and a controlling object that coordinates the activities of the DLL's other components. In addition, if an object is a link, the DLL also includes a linking component, or *linked object,* which keeps track of the name and location of the link source.

The *cache* contains data and presentation information sufficient for the handler to display a loaded, but not running, object in its container. OLE provides an implementation of the cache used by OLE's default object handler and the link object. The cache stores data in formats needed by the object handler to satisfy container draw requests. When an object's data changes, the object sends a notification to the cache so that an update can occur. For more information on the cache, see "View Caching" later in this chapter.

The Default Handler and Custom Handlers

The default handler, an implementation provided by OLE, is used by most applications as the handler. An application implements a custom handler when the default handler's capabilities are insufficient. A custom handler can either completely replace the default handler or use parts of the functionality it provides where appropriate. In the latter case, the application handler is implemented as an aggregate object composed of a new control object and the default handler. Combination application/default handlers are also known as *in-process handlers*. The *remoting handler* is used for objects that are not assigned a CLSID in the system registry or that have no specified handler. All that is required from a handler for these types of objects is that they pass information across the process boundary.

In-Process Servers

If you implement an OLE server application as an in-process server – a DLL running in the process space of the container application – rather than as a local server – an EXE running in its own process space – communication between container and server is simplified because communication between the two can take the form of normal function calls. Remote procedure calls are not required because the two applications run in the same process space. As you would expect, the objects that manage the marshaling of parameters are also unnecessary, although they may be aggregated within the DLL without interfering with the communication between container and server.

When an OLE server application is implemented as an in-process server, a separate object handler is not required because the server itself lives in the client's process space. The main difference between an in-process server and object handler is that the server is able to manage the object in a running state while the handler cannot. One consequence of this difference is that a server must provide a user interface for manipulating the running object, while a handler delegates this requirement to the object's server. In creating an in-process server, you can aggregate on the OLE default handler, letting it handle basic chores, such as display, storage, and notifications while you implement only those services that the handler either does not provide or does not implement in the way you require.

Advantages

The advantages of implementing your application as an in-process server are speed and combining some the advantages of an object handler and a local server. In-process servers are faster than local servers for several reasons. First, because they are smaller and run in the process space of the container application, they load more quickly. Second, they are optimized to perform certain tasks. Third, communication between container and server does not rely on remote procedure calls.

Disadvantages

In-process servers provide the speed and size advantage of an object handler with the editing capability of a local server. So why would you ever choose to implement your OLE application as a local server rather than an in-process server? There are several reasons:

- Security. Only a local server has its address space isolated from that of the client. An in-process server shares the address space and process context of the client and can therefore be less robust in the face of faults or malicious programming.
- Granularity. A local server can host multiple instances of its object across many different clients, sharing server state between objects in multiple clients in ways that would be difficult or impossible if implemented as an in-process server, which is simply a DLL loaded into each client.
- Compatibility. If you choose to implement an in-process server, you relinquish compatibility with OLE 1, which does not support such servers. This will not be a consideration for many developers, but if it is, then it is of critical concern.
- Inability to support links. An in-process server cannot serve as a link source. Since a DLL cannot run by itself, it cannot create a file object to be linked to.

Despite these disadvantages, an in-process server can be an excellent choice for its speed and size – if it fits all your other requirements.

Linked Objects

Linked objects, like embedded objects, rely on an object handler to communicate with server applications.The linked object itself, however, manages the naming and tracking of link sources.The linked object acts like an in-process server. For example, when activated, a linked object locates and launches the OLE server application that is the link source.

A linked object's handler is made up of two main components: the handler component and the linking component. The handler component contains the controlling and remoting pieces and functions much like a handler for an embedded object. The linking component has its own controller and cache and provides access to the object's structured storage. The linking component's controller supports source naming through the use of monikers, and binding, the process of locating and running the link source. (For more information on monikers and binding, see "The Component Object Model.")

When a user initially creates a linked object or loads an existing one from storage, the container loads an instance of the linking component into memory, along with the object handler. The linking component supplies interfaces – most notably **IOleLink** – that identify the object as a link and enable it to manage the naming, tracking, and updating of its link source.

By implementing the **IOleLink** interface, a linked object provides its container with functions that support linking. Only linked objects implement **IOleLink**, and by querying for this interface a container can determine whether a given object is embedded or linked. The most important function provided by **IOleLink** enables a container to binding to the source of the linked object, that is, to activate the connection to the document that stores the linked object's native data. **IOleLink** also defines functions for managing information about the linked object, such as cached presentation data and the location of the link source.

When a compound document containing a linked object is saved, the link's data is saved with the link source, not with the container. Only information about its name and location is saved along with the compound document. This behavior is in contrast to that of an embedded object, whose data is stored along with that of its container.

Container applications can provide information about their embedded objects such that the latter, or portions thereof, can act as link sources. By implementing support for linking to your container's embedded objects, you make nested embeddings possible, relieving the user of having to track down the originals of every embedding object to which a link is desired. For example, if a user wants to embed a Microsoft® Excel worksheet in Microsoft® Word, and the worksheet contains a bitmap created in Paintbrush™, the user can link to a copy of the bitmap contained in the worksheet rather than the original.

Notifications

Notifications are callbacks generated by an object when it detects a change in its name, state, data, or presentation. Containers and other clients require notifications to respond appropriately to these changes. A container registers to receive notifications by setting up an advisory connection to an object of interest. Other interested clients can do the same. The container also creates an advisory sink to receive the notifications. Using the connection established by the container, an object experiencing a change notifies the advisory sink. Upon receiving a notification, the container takes whatever action has been defined for the type of change that has occurred.

Types of Notifications

Notifications fall into three groups: compound document, data, and view. An object sends compound document notifications in response to being renamed, saved, closed or, in the case of a link, having its link source renamed. As you would expect, objects send data notifications in response to changes in their data and send view notifications in response to changes in their presentation. Container applications must register separately for each of these notification types, but all can be handled by a single advisory sink.

All containers, the object handler, and the linked object register for compound document notifications. The typical container also registers for view notifications. Data notifications are usually sent to both the linked object and object handler. A special purpose container, such as one that renders the data itself, might benefit from receiving data notifications instead of view notifications. For example, an embedded chart container with a link to a table can register for data notifications. Because a change to the table affects the chart, the receipt of a data notification would direct the container to get the new tabular data.

How Notifications Work

Notifications originate in the object application and flow to the container by way of the object handler. If the object is a linked object, the linked object intercepts the notifications from the object handler and notifies the container directly.

An object application must manage registration requests, keeping track of where to send which notifications and sending those notifications when appropriate. OLE provides two component objects to simplify this task: the OleAdviseHolder for compound document notifications and the DataAdviseHolder for data notifications.

Object applications determine the conditions that prompt the sending of each specific notification and the frequency with which each notification should be sent. When it is appropriate for multiple notifications to be sent, it does not matter which notification is sent first; they can be sent in any order.

The timing of notifications affects the performance and coordination between an object application and its containers. Whereas notifications sent too frequently slow processing, notifications sent too infrequently result in an out-of-sync container. Notification frequency can be compared with the rate at which an application repaints. Therefore, using similar logic for the timing of notifications (as is used for repainting) is wise.

Compound Document Interfaces

The following tables list the interfaces implemented by OLE containers, OLE servers, and compound document objects. The required interfaces *must* be implemented on the components for which they are listed. For example, containers must implement the **IOleClientSite** and **IAdviseSink** interfaces. All other features are optional. If you want to include a particular feature in your application, however, you must implement the interfaces shown for that feature in the table below. All other interfaces are required only if you are including a particular feature. For example, if you want your application to do message filtering (recommended), you must implement **IMessageFilter**.

The following table lists required and optional behaviors for OLE containers and which interfaces you must implement for each.

OLE Containers

Behavior	Interfaces
Required Behaviors	IOleClientSite
	IAdviseSink
Message Filtering	IMessageFilter
Linking	none
Linking to Embedded Objects	IOleItemContainer
	IPersistFile
	IClassFactory
In-Place Activation	IOleInPlaceSite
	IOleInPlaceActive-Frame
	IOleInPlaceUIObject
Drag and Drop	IDropSource
	IDropTarget
	IDataObject

The following table lists required and optional behaviors for OLE servers and their compound document objects and which interfaces you must implement for each. The table distinguishes OLE servers and their objects in order to clarify which component implements which interfaces. The table also notes the different requirements of objects provided by out-of-process versus in-process servers.

Feature	OLE Server	Compound Document Object	
		Out-of-Process	**In-Process**
Required Behaviors	IClassFactory	IOleObject	IOleObject
		IDataObject	IDataObject
		IPersistStorage	IPersistStorage
			IViewObject2
			IOleCache2
Message Filtering	IMessageFilter		
Linking	IOleItemContainer		IOleLink
	IPersistFile		IExternalConnection
In-Place-Activation		IOleInPlaceObject	IOleInPlaceObject

		IOleInPlace-ActiveObject	IOleInPlaceActive-Object
Drag and Drop	IDropSource IDropTarget IDataObject		

Object States

A compound object exists in one of three states: passive, loaded, or running. A compound-document object's state describes the relationship between the object in its container and the application responsible for its creation. The following table summarizes these states.

Object State	Description
Passive	The compound-document object exists only in storage, either on disk or in a database. In this state, the object is unavailable for viewing or editing.
Loaded	The object's data structures created by the object handler are in the container's memory. The container has established communication with the object handler and there is cached presentation data available for rendering the object. Calls are processed by the object handler. This state, because of its low overhead, is used when a user is simply viewing or printing an object.
Running	The objects that control remoting have been created and the OLE server application is running. The object's interfaces are accessible, and the container can receive notification of changes. In this state, an end user can edit or otherwise manipulate the object.

Entering the Loaded State

When an object enters the loaded state, the in-memory structures representing the object are created so that operations can be invoked on it. The object's handler or in-process server is loaded. This process, referred to as *instantiation,* occurs when an object is loaded from persistent storage (a transition from the passive to the loaded state) or when an object is being created for the first time.

Internally, instantiation is a two-phase process. An object of the appropriate class is created, after which a method on that object is called to perform initialization and give access to the object's data. The initialization method is defined in one of the object's supported interfaces. The particular initialization method called depends on the context in which the object is being instantiated and the location of the initialization data.

Entering the Running State

When an embedded object makes the transition to the running state, the object handler must locate and run the server application in order to utilize the services that only the server provides. Embedded objects are placed in the running state either explicitly through a request by the container, such as a need to draw a format not currently cached, or implicitly by OLE in response to invoking some operation, such as when a user of the container double-clicks the object.

When a linked object makes the transition into the running state, the process is known as *binding*. In the process of binding, the object handler asks its stored moniker to locate the link's data, then runs the server application.

At first glance, binding a linked object appears to be no more complicated than running an embedded object. However, the following points complicate the process:

- A link can refer to an object, or a portion thereof, that is embedded in another container. This capability implies a potential for nested embeddings. Resolving references to such a hierarchy requires recursively traversing a *composite moniker*, beginning with the rightmost member.
- When the link source is running, OLE binds to the running instance of the object rather than running another instance. In the case of nested embedded objects, one of which is the link source, OLE must be able to bind to an already running object at any point.
- Running an object requires accessing the storage area for the object. When an embedded object is run, OLE receives a pointer to the storage during the load process, which it passes on to the OLE server application. For linked objects, however, there is no standard interface for accessing storage. The OLE server application may use the file system interface or some other mechanism.

Entering the Passive State

Object closure forces an embedded or linked object into the passive state. It is typically initiated from the OLE server application's user interface, such as when the user selects the File Close command. In this case, the OLE server application notifies the container, which releases its reference count on the object. When all references to the object have been released, the object can be freed. When all objects have been freed, the OLE server application can safely terminate.

A container application can also initiate object closure. To close an object, the container releases its reference count after completing an optional save operation. You can design containers to release objects when they are deactivating after an in-place activation session, allowing the user to click outside the object without losing the active editing session.

Implementing In-Place Activation

In-place activation enables a user to interact with an embedded object without leaving the container document. When a user activates the object, a *composite menu bar* comprising elements from both the container application's and server application's menu bars replaces the container's main menu bar. Commands and features from both applications are thus available to the user, including context sensitive help for the active object. When a user begins working with some non-object portion of the document, the object is deactivated, causing the container document's original menu to replace the composite menu.

This capability originally went by the name of *in-place editing.* The name was changed because editing is only one way for a user to interact with a running object. Sound clips, for example, can be listened to instead of editing. Video clips can be viewed instead of editing. In-place activation is particularly apt in the case of video clips because it allows them to run in place, without calling up a separate window. This could be critical if the video were to be viewed, say, in conjunction with adjacent text data in the container document.

Implementing in-place activation is strictly optional for both container and server applications. OLE still supports the model in which activating an object causes the server application to open a separate window. Linked objects always open in a separate window to emphasize that they reside in a separate document.

In-place activation begins with the object in response to an **IOleObject::DoVerb** call from its container. This call usually happens in response to a user double-clicking the object or selecting a command (verb) from the container application's Edit menu.

The in-place window receives keyboard and mouse input while the embedded object is active. When a user selects commands from the composite menu bar, the command and associated menu messages are sent to the container or object application, depending on which owns the particular drop-down menu selected. Input by means of an object's rulers, toolbars, or frame adornments go directly to the embedded object, which owns these windows.

An in-place-activated embedded object remains active until either the container deactivates it in response to user input or the object voluntarily gives up the active state, as a video clip might do, for example. A user can deactivate an object by clicking inside the container document but outside the object's in-place-activation window, or simply by clicking another object. An in-place-activated object remains active, however, if the user clicks the container's title bar, scroll bar or, in particular, menu bar.

You can implement an in-place-activation-object server either as an in-process server (DLL) or a local server (EXE). In both cases, the composite menu bar contains items (typically drop-down menus) from both the server and container processes. In the case of a in-process server, the in-place activation window is simply another child window in the container's window hierarchy, receiving its input through the container application's message pump.

In the case of a local server, the in-place activation window belongs to the embedded object's server application process, but its parent window belongs to the container. Input for the in-place-activation window appears in the server's message queue and is dispatched by the server's message loop. The OLE libraries are responsible for seeing to it that menu commands and messages are dispatched correctly.

Creating Linked and Embedded Objects from Existing Data

A user typically assembles a compound document by using either the clipboard or drag and drop to copy a data object from its server application to the user's container application. With applications that support OLE, the user can initiate the transfer from either the server or the container. For example, the server can copy data to the clipboard in the server application, then switch to the container application and choose Paste Special/Embedded Object or an equivalent menu command to create a new embedded object from the selected data. Or, the user can drag the data from one application to the other. The process is similar for creating a linked object.

> **Note** An application that functions as both OLE server and container can use a selection of its own data to create an embedded or linked object at a new location within the same document.

Data transfer between OLE server and container applications is built on uniform data transfer, as described in Chapter 4, "Data Transfer." OLE servers and object handlers implement **IDataObject** in order to make their data available for transfers using either the clipboard or drag and drop. OLE objects support all the usual clipboard formats. In addition, they support six clipboard formats that support the creation of linked and embedded objects from a selected data object.

OLE clipboard formats describe data objects that, upon being dropped or pasted in OLE containers, are to become embedded or linked compound-document objects. The data object presents these formats to container applications in order of their fidelity as descriptions of the data. In other words, the object presents first the format that best represents it, followed by the next best format, and so on. This intentional ordering encourages a container application to use the best possible format.

View Caching

A container application must be able to obtain a presentation of an object for the purpose of displaying or printing it for users when the document is open but the object's server application is not running or is not installed on the user's machine. To assume, however, that the servers for all the objects that might conceivably be found in a document are installed on every user's machine and can always run on demand is unrealistic. The default object handler, which *is* available at all times, solves this dilemma by caching object presentations in the document's storage and manipulating these presentations on any platform regardless of the availablility of the object server on any particular installation of the container.

Containers can maintain drawing presentations for one or more specific target devices in addition to the one required to maintain the object on screen. Moreover, if you port the object from one platform to another, OLE automatically converts the object's data formats to ones supported on the new platform. For example, if you move an object from Windows to the Macintosh, OLE will convert its metafile presentations to PICT formats.

In order to present an accurate representation of an embedded object to the user, the object's container application initiates a dialog with the object handler, requesting data and drawing instructions. To be able to fulfill the container's requests, the handler must implement the **IDataObject**, **IViewObject2**, and **IOleCache** interfaces.

IDataObject enables an OLE container application to get data from and send data to its embedded or linked objects. When data changes in an object, this interface provides a way for the object to make its new data available to its container and provides the container with a way to update the data in its copy of the object. (For a discussion of data transfer in general, see Chapter 4, "Data Transfer.")

The **IViewObject2** interface is very much like the **IDataObject** interface except that it asks an object to draw itself on a device context, such as a screen, printer, or metafile, rather than move or copy its data to memory or some other transfer medium. The purpose of the interface is to enable an OLE container to obtain alternative pictorial representations of its embedded objects, whose data it already has, thereby avoiding the overhead of having to transfer entirely new instances of the same data objects simply to obtain new drawing instructions. Instead, the **IViewObject2** interface enables the container to ask an object to provide a pictorial representation of itself by drawing on a device context specified by the container.

When calling the **IViewObject2** interface, a container application can also specify that the object draw itself on a target device different than the one on which it will actually be rendered. This enables the container, as needed, to generate different renderings from a single object. For example, the caller could ask the object to compose itself for a printer even though the resulting drawing will be rendered on screen. The result, of course, would be a print-preview of the object.

The **IViewObject2** interface also provides methods that enable containers to register for view-change notifications. As with data and OLE advisories, a view advisory connection enables a container to update its renderings of an object at its own convenience rather than in response to a call from the object. For example, if a new version of an object's server application were to offer additional views of the same data, the object's default handler would call each container's implementation of **IAdviseSink::OnViewChange** to let them know that the new presentations were available. The container would retrieve this information from the advise sink only when needed.

Because Windows device contexts have meaning only within a single process, you cannot pass **IViewObject2** pointers across process boundaries. As a result, OLE local and remote servers have no need whatsoever to implement the interface, which wouldn't work properly even if they did. Only object handlers and in-process servers implement the **IViewObject2** interface. OLE provides a default implementation, which you can use in your own OLE in-process servers and object handlers simply by aggregating the OLE default handler. Or you can write your own implementation of **IViewObject**.

An object implements the **IOleCache** interface in order to let the handler know what capabilities it should cache. The object handler also owns the cache and ensures it is kept up to date. As the embedded object enters the running state, the handler sets up appropriate advisory connections on the server object, with itself acting as the sink. The **IDataObject** and **IViewObject2** interface implementations operate out of data cached on the client side. The handler's implementation of **IViewObject2** is responsible for determining what data formats to cache in order to satisfy client draw requests. The handler's implementation of **IDataObject** is responsible for getting data in various formats, etc., between memory and the underlying **IStorage** instance of the embedded object. Custom handlers can use these implementations by aggregating on the default handler.

Note The **IViewObject2** interface is a simple functional extension of **IViewObject** and should be implemented instead of the latter interface, which is now obsolete. In addition to providing the **IViewObject** methods, the **IViewObject2** interface provides a single additional member, **GetExtent**, which enables a container application to get the size of an object's presentation from the cache without first having to move the object into the running state with a call to **IOleObject::GetExtent**.

ActiveX Controls

ActiveX controls technology rests on a foundation consisting of COM, connectable objects, compound documents, property pages, OLE automation, object persistence, and system-provided font and picture objects. As summarized below, each of these core technologies plays a role in controls:

COM
A control is essentially a COM object that exposes the **IUnknown** interface, through which clients can obtain pointers to its other interfaces. Controls can support licensing through **IClassFactory2** and self-registration. See The Component Object Model chapter for more information on COM, licensing, and self-registration.

Connectable objects
Controls can support outgoing interfaces through connectable objects so that the control can communicate with its client. For example, an outgoing interface can trigger an action in the client, can notify the client of some change in the control, or can request permission from the client before the control takes some action. See the Connectable Objects chapter for more information on how connectable objects work.

Uniform data transfer
Controls can support being dragged and dropped within a container with help from their container. See **IOleInPlaceObjectWindowless::GetDropTarget** for more information on drag and drop.

Compound documents
A control can be an in-place active object that can be embedded in a containing client. An end-user activates the control to initiate an action in the container application. See the Compound Documents chapter for more information on in-place activation and other compound document interfaces.

Property pages
Controls can provide property pages so end users can view and change the control's properties. See the Property Pages and Property Sheets chapter for more information on how property pages work.

OLE automation
Controls can provide programmability through OLE automation so clients can take advantage of the control's features through a programming language supplied by the client. See the OLE Automation section for more information on OLE automation.

Persistent storage
A control can implement one or more of several persistence interfaces to support persistence of its state. The control implementer must decide what kinds of persistence are most important and implement the appropriate persistence interfaces. The client decides which interface it prefers to use. See The Component Object Model chapter for more information on all the persistence interfaces.

Font and picture objects
Controls can use these system provided objects to provide a visual representation of themselves within the client. The font object implements several interfaces, including **IFont** and **IFontDisp**. A font object can be created with **OleCreateFontIndirect**. The picture object also implements several interfaces, including **IPicture** and **IPictureDisp**. A picture object can be created using **OleCreatePictureIndirect** and can loaded from a stream with **OleLoadPicture**. The standard font and picture objects are described in this chapter.

It is important to understand that these features can be used in any OLE object. One does not need to implement a control in order to use these features. Also, the only required interface on a control is **IUnknown**. The control optionally supports other interfaces based on the need to support the related features.

In addition to these features, the following interfaces and API functions are specific to controls technology: **IOleControl**, **IOleControlSite**, **ISimpleFrameSite**, the API function **OleTranslateColor**. Also specific to controls are a set of standards for properties and methods that a control or a control container can support.

See the ActiveX Control and Control Container Guidelines appendix for more information on controls and their containers.

Notes For new systems (NT 4.0 and above), the system library OLEAUT32.DLL contains implementations of the API functions (OleCreatePropertyFrame, OleCreatePropertyFrameIndirect, OleCreateFontIndirect, OleCreatePictureIndirect, OleIconToPicture, OleLoadPicture, and OleTranslateColor).

In addition, OLEAUT32.DLL contains the implementations of the standard font and picture objects, as well as a type library for all the interfaces used with controls as well as the additional data structures and data types.

For older systems, the redistributable library, OLEPRO32.DLL, contains these implementations. Applications or components for older systems that use these API functions can link with the import library, OLEPRO32.LIB.

ActiveX Controls Architecture

As noted above, ActiveX controls technology builds on a foundation of many lower-level objects and interfaces in OLE. The exact interfaces available on a control vary with its capabilities. This section takes a closer look at the capabilities a control might provide.

Controls are used to provide the building blocks for creating user interfaces in applications. For example, a button that initiates some action in the container application when it is clicked is a simple control. The following aspects are involved in providing these user interface building blocks:

- A control can be embedded within its container client to support some user interface activity within the client. Thus, a control needs to provide a visual representation of itself when it is embedded within the container and needs to provide a way to save its state, for example, its property values and its position within its container. The client must support being a container with objects embedded in it.
- By activating the control using a keyboard or mouse, the end user initiates some action in the client application. Thus, a control must respond to keyboard activity and must be able to communicate with its client so it can notify its container of its activities and trigger events in the client.
- The client also typically provides a programming language through which the end user can initiate actions provided by the control's properties and methods. Thus, a control must support automation and some set of design-time versus run-time features as well.

As a result of its role in providing user interface building blocks, a control typically supports features in the following areas using OLE technologies as indicated:

Properties and methods
Like any OLE object, a control can provide much of its functionality through a set of incoming interfaces with properties and methods. The container can supply additional ambient properties, and it can support extending the control's properties through aggregation. These features rest on OLE automation, property pages, connectable objects, and ActiveX control technologies.

Events
In addition to providing properties and methods, an ActiveX control can also provide outgoing interfaces to notify its client of events. The client must support handling of these events. These features use OLE automation and connectable objects.

Visual representation
A control can support positioning and displaying itself within its container. The container positions the control and determines its size. These features use compound document technology, including OLE drag and drop technology.

Keyboard handling
A control can respond to keyboard accelerators so the end-user can initiate actions performed by the control. The container manages keyboard activity for all its embedded controls. These features use control and compound document technologies.

Persistence
A control can save its state. The client manages the persistence of its embedded controls. These features use structured storage and object persistence technologies.

Registration and licensing
A control typically supports self-registration and creates a set of registry entries when it is instantiated. A control can also be licensed to prevent unauthorized use.

Most of these features involve both the control and its client container.

The following sections describe design considerations in each of these areas and describe how the OLE technologies mentioned previously are used in controls to support these areas of features.

ActiveX Controls Interfaces

In addition to other mechanisms for communicating between the control and its client, ActiveX controls technology specifies the **IOleControl** and **IOleControlSite** interfaces for client-control communication. There is also the **ISimpleFrameSite** interface for simple control containers.

These three interfaces are, however, specific to controls are not generally useful outside the context of controls. These interfaces are defined as follows:

```
interface IOleControl : IUnknown
    {
    HRESULT GetControlInfo([out] CONTROLINFO *pCI);
    HRESULT OnMnemonic([in] LPMSG pMsg);
    HRESULT OnAmbientPropertyChange([in] DISPID dispID);
    HRESULT FreezeEvents([in] BOOL bFreeze);
    }

interface IOleControlSite : IUnknown
    {
    HRESULT OnControlInfoChanged(void);
    HRESULT LockInPlaceActive([in] BOOL fLock);
    HRESULT GetExtendedControl([out] IDispatch **ppDisp);
    HRESULT TransformCoords([in-out] POINTL *pptlHimetric
        , [in-out] POINTF *pptfContainer, [in] DWORD dwFlags);
    HRESULT TranslateAccelerator([in] LPMSG pMsg
        , [in] DWORD grfModifiers);
    HRESULT OnFocus([in] BOOL fGotFocus);
    HRESULT ShowPropertyFrame(void);
    }

interface ISimpleFrameSite : IUnknown
    {
    HRESULT PreMessageFilter([in] HWND hWnd, [in] UINT msg
        , [in] WPARAM wp, [in] LPARAM lp, [out] LRESULT *plResult
        , [out] DWORD *pdwCookie);
    HRESULT PostMessageFilter([in] HWND hWnd, [in] UINT msg
        , [in] WPARAM wp, [in] LPARAM lp, [out] LRESULT *plResult
        , [in] DWORD dwCookie);
    }
```

Some controls, like a group box, are merely a simple container of other controls. In such cases, the simple control, called a simple frame, doesn't have to implement all the container requirements. It can delegate most of the interface calls from its contained controls to the container that manages the simple frame. Besides interface calls, the simple frame also has to deal with Windows messages that potentially come from controls within it. For this reason, a container supplies **ISimpleFrameSite** to allow such simple frame controls to pass messages up to the contain. **PreMessageFilter** gives the container first crack at the message; **PostMessageFilter** is called after the simple frame has process the message itself.

IOleControl and **IOleControlSite** are described throughout the following sections.

Properties and Methods

Like any OLE object, a control provides much of its functionality through a set of incoming interfaces with properties and methods.

A control exposes properties and methods through OLE automation so that containers can access them under the control of a container-supplied programming language.

To support access to properties through a user interface, a control provides property pages, support for OLEIVERB_PROPERTIES, per property browsing, and data binding through property change notfications.

- Through property pages a control can display its properties, independent of its container, if necessary.
- By supporting OLEIVERB_PROPERTIES, the Properties item is displayed on the container's menu. Then, the end user can select the Properties item to view the control's property pages and modify the properties.
- Per property browsing supports a container that can display the control's properties as part of a larger property sheet that may include properties from several controls in the container.
- Through property change notification, a control can notify a client that its properties have change, allowing the client to take any necessary actions as a result.

See the Property Pages and Property Sheets chapter for more information on these features.

Control Properties

In addition to properties defined and implemented by the control itself, ActiveX controls technology also involves:

Ambient properties

These are exposed by the container through a control client site to provide environmental values that apply to all controls embedded in the container. For example, a container can provide a default background color or a default font that the control can use. Ambient properties are exposed through **IDispatch** implemented on a container's site object.

The container calls the control's **IOleControl::OnAmbientPropertyChange** method when any of its ambient properties change value. In response, a control may need to update its own internal or visual state in response. The container indicates which ambient property changed with the DISPID parameter or may pass DISPID_UNKNOWN to indicate that multiple ambient properties changed.

Extended properties

These are actually implemented by a container to wrap the controls it contains to provide container-managed properties that appear as if they were native control properties. The container can aggregate the control, adding the extended properties to supplement or override the control's properties.

The aggregated object is called an extended control. To the container, the extended control appears as the control itself and extended properties appear to be exposed by the control. The container supports an extended control through its client site method **IOleControlSite::GetExtendedControl**. The **GetExtendedControl** method allows controls to navigate through the site to the extended control object provided for them by the container, if the container supports this feature.

A container can also choose to show property pages for its extended controls in addition to those pages that a control would normally specify through **ISpecifyPropertyPages**. Because of this, a control has to ask a container to show a property frame before the control attempts to do so itself. The control calls **IOleControlSite::ShowPropertyFrame** to do this. If the container implements this function then it shows the property frame itself; if the method returns an error then the control can show the property frame.

Standard Properties

OLE defines a set of standard DISPIDs for all three kinds of properties: control, ambient, and extended. The following tables list these standards for control properties, ambient properties, and extended properties.

Control Property	Type	Description
BackColor, ForeColor, FillColor, BorderColor	OLE_COLOR	The control's color scheme
BackStyle, FillStyle, BorderStyle, BorderWidth, BorderVisible, DrawStyle, DrawWidth	short or long	Bits that define a control's visual behavior, such as being solid or transparent, having thick or thin borders, line styles, and so forth.
Font	IDispatch *	The font used in the control, which is an **IDispatch** pointer to a standard font object. See Standard Font Object below for more information.
Caption, Text	BSTR	Strings containing the control's label (the caption) or its textual contents (the text). Note that the caption does not necessarily name the control in the container. See the extended Name property in the following table.
Enabled	BOOL	Determines if the control is enabled or disabled. If disabled, the control is probably grayed.
Window	HWND	The window handle of the control, if it has one.
TabStop	BOOL	Determines if this control is a tab stop.

Ambient Property	Type	Description
BackColor, ForeColor	OLE_COLOR	Provides controls with the default background and foreground colors. Use by a control is optional.
Font	IDispatch *	A pointer to a standard font object that defines the default font for the form. Use by a control is optional. See Standard Font Object below for more information.
LocaleID	LCID	The language used in the container. Use by a control is recommended.
UserMode	BOOL	Describes whether the container is in a design mode (FALSE) or run-mode (TRUE), which a control should use to change its available functionality as necessary.
UIDead	BOOL	Describes whether the container is in a mode where controls should ignore user input. This applies irrespective of UserMode. A container might always set UIDead to TRUE in design mode,

		and may set it to true when it has hit a breakpoint or such during run mode. A control must pay attention to this property.
MessageReflect	BOOL	Specifies whether the container would like to receive Windows messages such as WM_CTLCOLOR, WM_DRAWITEM, WM_PARENTNOTIFY, and so on as events.
SupportsMnemonics	BOOL	Describes whether the container processes mnemonics or not. A control can do whatever it wants with this information, such as not underline characters it would normally use as a mnemonic.
ShowGrabHandles, ShowHatching	BOOL	Describes whether a control should show a hatch border or grab handles (in the hatch border) when in-place active. Controls must obey these properties, giving the container ultimate control over who actually draws these bits of user interface. A control container may want to draw its own instead of relying on each control, in which case these ambients will always be FALSE.
DisplayAsDefault	BOOL	The container will expose a TRUE for this property through whatever site contains what is marked as the default button when the button control should draw itself with a thicker default frame.

Extended Property	**Type**	**Description**
Name	BSTR	The container's name for the control.
Visible	BOOL	The control's visibility.
Parent	IDispatch *	The dispinterface of the form containing the control.
Default, Cancel	BOOL	Indicates if this control is the default or cancel button.

All of these standard properties have negative DISPID values, indicating their standard status.

Note that to avoid conflicts in the programmatic symbols for these DISPIDs, all ambient properties are given symbols in the form DISPID_AMBIENT_*property* as in DISPID_AMBIENT_FORECOLOR. All other symbols use DISPID_*property* as usual.

Some ambient properties, as well as control properties, involve colors. Controls deal with color in a slightly different way than normal Win32 APIs do. The OLE_COLOR type mentioned in the previous tables can refer to a Win32 standard COLORREF type, an index to a palette, a palette-relative index, or a system color index used with the Win32 API function **GetSysColor**. The OLE API function **OleTranslateColor** converts an OLE_COLOR type to a COLORREF type given a palette.

Standard Font Object

The standard ambient font property supplied by the container and the standard font property supplied by the control both provide a standard font object. That is, these standard fonts supply an **IDispatch** pointer to a standard font object.

The font object is a system-provided implementation of a set of interfaces on top of the underlying GDI font support. A font object is created through the API function **OleCreateFontIndirect** given a **FONTDESC** structure. The font object supports a number of read-write properties as well as custom methods through its interface **IFont**, and supports the same set of properties (but not the methods) through a dispinterface **IFontDisp**. The dispinterface is used for the font properties mentioned previously. The properties correspond to the GDI font attributes that are described in the **LOGFONT** structure.

The font object also supports the outgoing interface **IPropertyNotifySink** so that a client can determine when font properties change. Since the font object supports at least one outgoing interface, it also implements **IConnectionPointContainer** and one connection point for **IPropertyNotifySink** for this purpose.

The font object provides an hFont property that is a Windows font handle that conforms to the other attributes specified for the font. The font object delays realizing this GDI hFont when possible, so consecutively setting two properties on a font won't cause an intermediate font to be realized. In addition, as an optimization, the standard font object maintains a cache of font handles. Two font objects in the same process that have identical properties will return the same font handle. The font object can remove fonts from this cache at will, which introduces special considerations for clients using this hFont property. See **IFont::get_hFont** for more details.

The font object also supports **IPersistStream** such that it can save and load itself from an instance of **IStream**. Any other object that uses a font object internally would normally save and load the font as part of the object's own persistence handling.

In addition, the font object supports **IDataObject** through which it provides a property set containing typed values for each font property.

Standard Picture Object

The standard picture object provides a language-neutral abstraction for GDI images: bitmaps, icons, metafiles, and enhanced metafiles. As with the standard font object, the system provides a standard implementation of this object. Its primary interfaces are **IPicture** and **IPictureDisp**, the latter being derived from **IDispatch** to provide access to the picture's properties through OLE automation. A picture object is created new with **OleCreatePictureIndirect**.

The picture object also supports the outgoing interface **IPropertyNotifySink** so that a client can determine when picture properties change. Accordingly, the picture object also supports **IConnectionPointContainer** and one connection point for **IPropertyNotifySink**.

The picture object also supports **IPersistStream** such that it can save and load itself from an instance of **IStream**. An object that uses a picture object internally would normally save and load the picture as part of the object's own persistence handling. The API function **OleLoadPicture** simplifies the creation of a picture object based on stream contents.

Control Methods

There are three standard methods that controls can support: **Refresh**, **DoClick**, and **AboutBox**. All of these standard methods have negative DISPID values, indicating their standard status.

Control Events

In addition to providing properties and methods, a control also provides outgoing interfaces to notify its client of events. The client must support handling of these events. See the Connectable Objects chapter for more information on how connectable objects work.

A control can support different outgoing interfaces for different purposes. All outgoing interfaces are marked as **source** interfaces in the control's type information, but only one is marked **default** to indicate that it is the primary outgoing interface.

A container can support one or more of the outgoing interfaces defined by a control. The control should be prepared to deal with containers that only provide support for some of their outgoing interfaces.

Controls support four kinds of events:

- Request events. A control requests permission from its client to do something by calling a method in the outgoing interface, thus triggering a request event. The client signals the control through a boolean, out-parameter in the method that the control called. The client can thus prevent the control from performing the action.
- Before events. A control notifies its client that it is going to do something by calling a method in the outgoing interface, thus triggering a before event. The client does not have the opportunity to prevent the action, but it can take any necessary steps given the action that is about to occur.
- After events. A control notifies its client that it has just done something by calling a method in the outgoing interface, thus triggering an after event. Again, the client cannot cancel this action, but it can take necessary steps given the action that has occurred.
- Do events. A control triggers a do event to allow its client to override the control's action and provide some alternative or supplementary actions. Usually, the method that a control calls for a do event has a number of parameters for negotiating with the client about the actions that will occur.

The following dispids are defined for standard events that controls can support: **Click**, **DblClick**, **KeyDown**, **KeyPress**, **KeyUp**, **MouseMove**, **MouseUp**, and **Error**. All of these standard events have negative DISPID values, indicating their standard status.

The **IOleControl::FreezeEvents** method, when called with TRUE, tells a control whether the container will bother handling events from the control until **FreezeEvents** is again called with FALSE. During this time control cannot depend on the container actually handling any events. If an event must be handled, the control should queue the event in order to fire it when FreezeEvents(FALSE) is called.

Visual Representation

A control supports positioning and displaying itself within its container through compound document technology and OLE drag and drop technology that involves both the control and its container. The control must be able to draw itself while the container manages the position of the control and its size.

Controls add to the basic functions provided by OLE documents. A control calls its client's **IOleClientSite::RequestNewObjectLayout** method to tell its container that it wants to change its size. The client calls the control's **IOleObject::GetExtent** to get the new size and calls **IOleInPlaceObject::SetObjectRects** to set the control to its new size.

Controls that support only **IPersistStream[Init]** do not support caching through **IOleCache2** because the cache requires support for **IPersistStorage**. However, these controls should provide a way for the client to render the control through **IDataObject::GetData** so the client can optionally create and manage its own cache of the presentation data for the control.

Controls use the HIMETRIC type for its coordinates. However, different containers can use different coordinate systems. The container wants to receive coordinates in its own system, but the control does not necessarily know what coordinates its container is using. To communicate successfully, the control needs a way to convert values to its container's coordinates. The container provides a site object with the **IOleControlSite::TransformCoords** method. The control calls this method on its container's client site first to convert its coordinates into the appropriate coordinates for the container. Then, it can pass the converted coordinates to the container.

Controls can call **IOleControlSite::LockInPlaceActive** in the container's site object to prevent the container from attempting to demote the control out of the in-place active state. Demoting the control in this way causes the control to be deactivated and its window destroyed, so if the control must maintain its window for a known duration it can call **LockInPlaceActive** to guarantee its state.

Keyboard Handling for Controls

A control responds to keyboard accelerators so the end-user can initiate actions performed by the control. The container manages keyboard activity for all its embedded controls. With compound documents, keyboard accelerators apply only to the currently active object. With controls, a mechanism has been added so that a control can respond to its keyboard mnemonics even if it is not currently UI-active.

The **IOleControl::GetControlInfo** and **IOleControl::OnMnemonic** methods and the **IOleControlSite::OnControlInfoChanged** method handle a control's keyboard mnemonics. A **CONTROLINFO** structure describes a control's mnemonic accelerators, and the flags passed back with it through the **IOleControl::GetControlInfo** method describe the control's behavior with the Enter and Esc keys. When a control changes its mnemonics, it calls **IOleControlSite::OnControlInfoChanged** so the container can reload the structure if necessary.

When a control is UI active, it is also the control with the focus. As controls are activated and deactivated between the in-place active and the UI active states, the control calls **IOleControlSite::OnFocus** to tell the container of such changes.

In addition, when a control is UI active, it will have first chance to process any keystrokes. To give a container the opportunity to process the keystroke before the control, the control calls **IOleControlSite::TranslateAccelerator**. If the container does not handle the keystroke, the control then processes it.

Persistence

A control implements one or more of several persistence interfaces to support persistence of its state. For example, the **IPersistStreamInit** interface supports stream-based persistence of the control's state. **IPersistStreamInit** is a replacement for **IPersistStream** and adds an initialization method, **InitNew**. The other methods are the same in both interfaces. **IPersistStreamInit** is not derived from **IPersistStream**; an object supports only one of the two interfaces based on whether it requires the ability to initialize new instances of itself.

Other persistence interfaces that a control can offer include: **IPersistStorage**, **IPersistMemory**, **IPersistPropertyBag**, **IPersistMoniker**. The control implementer must decide what kinds of persistence are most important and implement the appropriate persistence interfaces. The control implementer also decides what to save. For example, a control can save the current values of its properties or its location and size within its container. The client decides which interface it prefers to use.

Before loading a control from its persistent state, a client can check the OLEMISC_SETCLIENTSITEFIRST flag to determine if the control supports getting its client site and ambient properties before loading its persistent state. This optimization can save time when instantiating a control since the control is then free to ignore its persistent values rather than loading them only to have them overridden by ambient properties supplied by the client.

A control can also support saving and restoring its state in an OLE property set, a set of identifiers and values in a specified format. This feature can be useful with containers such as Visual Basic which saves its programs in a textual form. A control that wishes to support this feature implements **IDataObject::GetData** and **IDataObject::SetData** to pass its property values to and from the container, respectively. It is the container's job to convert this information to text and save it. The identifiers used by the control correspond to the control's property names and the values. See the OLE CDK for the definition of this property set.

Registration and Licensing

A control is usually provided as an in-process server (.DLL), although it can also be a local or remote server (.EXE).

A control typically supports self-registration and creates a set of registry entries when it is instantiated. A control can also be licensed to prevent unauthorized use. See The Component Object Model chapter for more information on self registration and licensing.

ActiveX Controls Registry Information

For 32-bit controls, there are a number of registry entries and flags that are used. Additionally, controls can support component categories to classify the features they provide.

Registry keys related to controls are italicized in the following list:

```
HKEY_CLASSES_ROOT
    CLSID
      {class id of control}
          ProgID = identifier
          InprocServer32 = <filename>.DLL
          DefaultIcon = <filename>.<ext>,resourceID
          ToolboxBitmap32 = <filename>.<ext>,resourceID
          Control
          verb
              n = &Properties...
          MiscStatus = 0
          TypeLib = {typelib ID for the object}
          Version = version number
```

The **DefaultIcon** entry is used to identify an icon to be displayed when the control is minimized to an icon. The Windows API function **ExtractIcon** is used to get the icon from the .DLL or .EXE file specified.

The **ToolboxBitmap32** entry identifies the module name and resource identifier for a 16*15 bitmap to use for the face of a toolbar or toolbox button. The standard Windows icon size is too large to be used for this purpose. This entry specifically supports control containers that have a design mode in which one selects controls and places them on a form being designed. For example, in Visual Basic, the control's icon is displayed in the Visual Basic toolbox during design mode.

The **Control** entry marks an object as a control. This entry is often used by containers to fill in dialog boxes. The container uses this sub-key to determine whether to include an object in a dialog box that displays controls.

The **Insertable** sub-key can also be used with controls, depending on whether the object can act only as an in-place embedded object with no special control features. Objects marked with **Insertable** appear in the Insert Object dialog box of their container. The **Insertable** entry generally means that the control has been tested with non-control containers.

Both the **Insertable** and the **Control** sub-keys are optional. A control can omit the **Insertable** sub-key if it not designed to work with older containers that do not understand controls. A control can omit the **Control** key if it is only designed to work with a specific container and thus does not wish to be inserted in other containers.

Controls should have a **Properties** verb, OLEIVERB_PROPERTIES, along with any other verbs they support. The **Properties** verb, as well as the standard verb OLEIVERB_PRIMARY, instructs the control to display its property sheet. The Properties verb is displayed as the Properties item on the container's menu when the control is active. This way, the control can display its own property page allowing some useful functionality to the end user, even if the container does not handle controls.

A control defines the **MiscStatus** key to describe itself to potential containers. The bits take on the values from **OLEMISC**, and controls add several values to this enumeration. See the **OLEMISC** enumeration values for more information. The client can obtain this information by calling **IOleObject::GetMiscStatus** without having to instantiate the control first.

Finally, **Version** describes the version of the control which should match the version of the type library

associated with this control.

Also in the type information for a control, the attribute **control** marks a **coclass** entry as describing a control.

Designing COM Interfaces

As an OLE developer, you implement and use objects that are based on the Component Object Model (COM). COM interfaces specify a contract between the interface implementor and its user. The contract applies to every COM object that supports that interface.

OLE provides a standard set of interfaces, but these OLE-provided interfaces may not be perfectly suited to your needs. Through custom interfaces, you can create new ones tailored to fulfill the specific needs of your application. Custom interfaces are extensions to the COM standard; they let you extend the standard behavior and still take advantage of the services provided by the base interface. Like all COM interfaces, custom interfaces derive from the **IUnknown** interface and must contain the three methods of **IUnknown**. Otherwise, they can support a wide range of methods and parameters. (Note that, currently, custom interfaces cannot include asynchronous methods. Asynchronous methods may be supported in future versions of OLE.)

Once the interface is written, anyone can use it. Calls to methods in COM interfaces can cross process boundaries as long as both processes are running on the same machine.

This chapter introduces custom interfaces and explains how to implement them. OLE experience is a prerequisite, as is a basic knowledge of RPC. Refer to **The Component Object Model** for basic information on COM required to design a COM interface, including threading models, and **COM Clients and Servers** for information on how clients and servers interact.

Custom interfaces require custom implementations of the **IMarshal** interface, so refer also to the section **Inter-object Communication** for a description of how this works. Refer to the *RPC Programmer's Guide and Reference* for information about the RPC programming environment.

Interface Design Rules

This section is a short summary of interface design rules. For details of custom interface design, refer to **Writing a Custom Interface**.

An object is not, by definition, a COM object unless it implements at least one interface. That interface must be **IUnknown** or an interface that is derived from **IUnknown**. In addition, the following rules apply to all interfaces implemented on a COM object:

- They must directly or indirectly inherit the methods of **IUnknown**.
- They must have a unique interface identifier.
- They must be immutable: once they are created and published, no part of their definition may change.
- All interface methods should have a return type of **HRESULT** so the portions of the system that handle remote processing can report RPC errors.
- All string parameters in interface methods should be Unicode.

Writing a Custom Interface

To write a custom interface that will be able to interact with COM objects, as discussed in the following section, you must obtain a Microsoft *Win32 Software Development Kit (SDK).* It contains a tool kit and all of the libraries you need for building a custom interface project. It provides more detailed information about how to write a custom interface using the MIDL compiler.

After installing the tool kit, you can design and write your custom interface using the MIDL compiler, and build the standard-marshaling proxy/stub DLL for your interface from the files generated by the compiler. You will then install the DLL in the system registry.

For details, see the following:

Designing Efficient Interfaces

Using a Custom Interface

Designing Efficient Interfaces

There are some special considerations to keep in mind when designing a custom interface. If you follow these guidelines, your marshaling code can be used across the network when support for remotable objects is provided in a future release.

First of all, because data is shipped across address spaces, using architecture-dependent types such as **int** may prohibit the data from being re-created correctly in the object application's address space. The size of an integer varies from architecture to architecture. Thus, the data type **int** should not be used for structure data members or interface method parameters. Specifying data types unambiguously to the MIDL compiler will allow your interface to be more easily transferred to remote machines with different architectures and addressing schemes. The MIDL compiler requires all integer variables that may be remoted to be explicitly declared as short, long, or their unsigned equivalents.

A frequently used data type for pointers, the **void *** construct, allows the code implementing a method to interpret the data pointed to according to the need. While local interfaces can use this construct, distributed applications cannot. This is because the MIDL compiler must know exactly what types of data are being transmitted so that the data can be accurately re-created on the receiving side. The **void *** construct is too vague.

Pointers to data must be used carefully. To re-create the data in the address space of the process that is called, RPC must know the exact size of the data. If, for example, a **char *** parameter points to a buffer of characters rather than to a single character (as is implied), the data cannot be correctly re-created. Use the syntax available with MIDL to accurately describe the data structures represented by your type definitions.

Initialization is essential for pointers that are embedded in arrays and structures and passed across process boundaries. Uninitialized pointers may work when passed to a program in the same process space, but proxies and stubs assume that all pointers are initialized with valid addresses or are null.

Be careful when aliasing pointers (allowing pointers to point to the same piece of memory). If the aliasing is intentional, these pointers should be declared aliased in the IDL file. Pointers declared as non-aliased should never alias each other.

The MIDL has directional attributes that let you specify the data flow direction. The proxies and stubs use these attributes to determine whether to send the data from the client to the object or from the object to the client, respectively. Data labeled as *out* only is uninitialized on the way to the interface stub; data labeled as *in* only does not affect the data structures upon return. The *in, out* attribute indicates that data is sent to the object initialized and the object will change it before sending it back.

It is also imperative to realize the importance of defining status codes. This must be coordinated by the definer of the interface to avoid conflicting error codes.

Finally, the designer of an interface must develop an understanding of how the interface will be used by client applications. In particular, the frequency of method calls across the interface boundary and the amount of data to be transferred to complete a given method call together determine whether the interface will be efficient across process and machine boundaries. Although OLE makes cross-process and, eventually, cross-network calls transparent to programs, it cannot make high-frequency and high-bandwidth calls efficient across address spaces. In some cases, it is more appropriate to design interfaces that will normally be implemented only as in-process servers while other interfaces are more appropriate for remote use.

For details, see the following:

- **Creating a Custom Interface**
- **Building a Proxy/Stub DLL**

- **Registering a Proxy/Stub DLL**

Creating a Custom Interface

Writing an application using a custom interface is similar to writing an OLE application using standard interfaces, with two important differences. First, in addition to registering the class identifier (CLSID) for your object, you also must register the unique interface identifier (IID) for your custom interface. Second, you must provide OLE with proxy and stub components capable of remoting the interface. In most cases, you will use the MIDL compiler to generate code to create the proxy/stub DLL for your custom interface. (For more information, see Files Generated for an OLE Interface in the Microsoft RPC documentation for the Win32 SDK.)

As an OLE developer, you should already be familiar with the concept of using CLSIDs and interface identifiers (IIDs) to uniquely identify class objects and interfaces. The MIDL compiler refers to universally unique identifiers (UUIDs) and uses UUIDs to uniquely represent its interfaces. In this case, a UUID is the same as an IID. Every custom interface must have a IID; the UUIDGEN.EXE tool provided with the toolkit can create one for you.

MIDL is a rich, complex language that allows you to define interface parameters carefully in terms of their direction and type. Although the MIDL compiler offers much, you will need only a small subset of its attributes to define your interface.

The IDL file consists of two parts: the interface header and the interface body. The interface header, delimited by brackets, contains information about the interface as a whole, such as its IID. A prerequisite to completing the interface header is running the RPC utility UUIDGEN.EXE to create the IID. Because IIDs must be unique, never reuse one of them by copying it from one IDL file to another. The header also contains the keyword **object**, indicating that this interface is a COM-style interface (that is, derived from **IUnknown**). Whereas standard RPC interfaces have the version attribute in their interface headers, object interfaces do not.

The following excerpt is from an IDL file:

```
[
    object,
    uuid(7ACC12C3-C4BB-101A-BB6E-0000C09A6549),
    pointer_default(unique)
]
```

The **uuid** attribute precedes the unique identifier for the interface. The **pointer_default** attribute specifies the default IDL type for all pointers except for those included in parameter lists. Parameter list pointers must be explicitly declared with the pointer attribute. The default pointer type may either be **unique**, **ref**, or **ptr**. For more information on using IDL with pointers, see the *RPC Programmer's Guide and Reference*.

The interface body contains declarations for data members, prototypes for all methods, and other information such as directives to the preprocessor and include statements for other IDL files. The following example is the interface body for **ICustomInterface**:

```
interface ICustomInterface : IUnknown
{
    import "unknwn.idl";

    HRESULT CustomReport(void);
}
```

The **import** attribute allows an interface to reference constructs defined in other IDL files. Because **ICustomInterface** derives from **IUnknown**, the UNKNWN.IDL file must be included. **ICustomInterface** has only one method that takes no parameters; few interfaces will be this simple.

Building a Proxy/Stub DLL

Completed IDL files must be compiled with the IDL compiler, MIDL.EXE. The IDL compiler takes each IDL file and generates proxy/stub code (IDLNAME_P.C), an interface header file (IDLNAME.H), and an interface identifier file (IDLNAME_I.C). The header files contain both C and C++ class definitions.

By default, MIDL.EXE generates names based on your IDL file's name. However, you can use the following command line switches to override the default:

Command line switch	Description
-header	Specifies the name of the interface header file.
-proxy	Specifies the name of the proxy source file.
-iid	Specifies the name of the interface identifier file.

After a successful compile of the IDL file, the generated files are run through the standard C/C++ compile and link steps. These source files implement helper functions for marshaling and an implementation of **DllGetClassObject** for the proxy/stub libraries, among other things. Note that the RPC4RT.LIB library includes an implementation of the **DllGetClassObject** function. In this provided implementation, the CLSID of the proxy/stub has to be the same as the IID of your custom interface. If you want to use a CLSID that differs from the IID then you must provide your own implementation of the **DllGetClassObject** function.

The following diagram shows all the pieces involved in a build of a custom proxy/stub DLL. The IDL file, ITF.IDL, is fed into the IDL compiler. Three files are generated: ITF_P.C, ITF.H, and ITF_I.C. These three files are compiled and linked and the result is PROXSTUB.DLL.

{ewc msdncd, EWGraphic, bsd23516 0 /a "SDK.WMF"}

Build Process for Proxy/Stub DLL.

It is important to remember that an IDL file is more than just a fancy header file for interfaces – it allows you to use your interfaces cross-machine, cross-process or even cross-thread.

IDL is a programming language for remoting. This means that all attributes should be verified before releasing IDL files for custom interfaces to customers.

Even if your interface will never be used out-of-process, it may be used cross-thread. The worst problem for an unchecked IDL file can arise for in-process servers that do not support multiple single-threaded apartments). A server that does not specify a threading model is implicitly single-threaded. Everything marked single threaded is forced over to the thread that first called **CoInitialize**or **CoInitializeEx**. If some other thread was the one that activated the object, all the interfaces on that single threaded server must be remoted back to the activating thread, which can result in a return of REGDB_E_IID_NOTREG in response to a call to QueryInterface). Unless you can absolutely assert that your interface is both in-process only, and always going to be called on the same thread, you will get remoted at some time.

Even if you create a proxy DLL, it is a good idea to test it by remoting your interfaces to avoid small errors. For example, " [in] char * pszString" means send a pointer to a single character. To send a pointer to a character string, you need to have "[in, string] char * pszString". Both generate the same header. Remoting your interfaces helps you avoid these problems.

Finally, with the advent of distributed OLE, it is crucial that IDL files are correct before you send them out, because the world is bigger and faster these days. If you made a mistake in your IDL, and the interface is not remoted correctly the first time you ship, then you are faced with the problem of trying to figure out

how to fix it, which can be very difficult. If you upgrade one machine to the correct proxy/stub, it won't interoperate with one that has the old proxy/stub. You either have to revise your interface with a new IID and leave the old one in for backwards compatibility, or you have to convert every client and every server machine everywhere at the same time.

The same considerations apply to those writing ODL files. IDL and ODL files just use different compilers to accomplish the same thing. In fact, with SUR and MIDL 3.0, the same tool does both, and the distinction vanishes (.IDL is to .ODL as CPP is to .CXX)

Registering a Proxy/Stub DLL

Before a client can use your custom interface, the proxy/stub DLL for the interface must be installed in the system registry. Registering the proxy/stub DLL involves creating a .REG file and running REGINI. In the following entries, notice that the IID for **ICustomInterface** is the same as the CLSID for the proxy/stub DLL.

```
\Registry\MACHINE\SOFTWARE\Classes\Interface\
    {7ACC12C3-C4BB-101A-BB6E-0000C09A6549}
        = ICustomInterface
\Registry\MACHINE\SOFTWARE\Classes\Interface\
    {7ACC12C3-C4BB-101A-BB6E-0000C09A6549}\ProxyStubClsid32
        = {7ACC12C3-C4BB-101A-BB6E-0000C09A6549}
\Registry\MACHINE\SOFTWARE\Classes\CLSID\
    {7ACC12C3-C4BB-101A-BB6E-0000C09A6549}
        = ICustomInterface_PSFactory
\Registry\MACHINE\SOFTWARE\Classes\CLSID\
    {7ACC12C3-C4BB-101A-BB6E-0000C09A6549}\InprocServer32
        = proxstub.dll
```

Using a Custom Interface

The client code is the user of the custom interface. To use any interface, custom or standard, a client must know its IID. In the following function, **CustomRpt**, the driver that calls **CustomRpt** passes it the name of the object that is converted to a wide-character format. The object name is fed to **CreateFileMoniker** so that a file moniker can be created and the client can bind to the running object. Once the object is running, **CustomRpt** can access a pointer to either an interface in the standard proxy/stub, such as **IPersistFile**, or to the custom interface, **ICustomInterface**.

```
void CustomRpt(char *pszObject)
{
    HRESULT          hr;
    WCHAR               szObject[128];
    WCHAR               wszMsg[128] = {L"Your Message Here...\n"};
    IMoniker            *pmkObject = NULL;
    IUnknown            *pIUnk = NULL;
    IPersistFile        *pIPersistFile = NULL;
    ICustomInterface    *pICustomInterface = NULL;

    // Create a wide-character version of the object's file name.
    wsprintf(wszObject, L"%hs", pszObject);

    // Get a file moniker for the object (a *.smp file).
    hr = CreateFileMoniker(wszObject, &pmkObject);

    if(FAILED(hr))
    {
        printf("Client: CreateFileMoniker for Object failed");
        return;
    }

    // BindMoniker is equivalent to calling CreateBindCtx() followed by
    // a call to BindToObject(). It has the net result of binding the
    // interface (specified by the IID) to the moniker.

    hr = BindMoniker(pmkObject, 0, IID_IUnknown, (void **)&pIUnk);
    if (FAILED(hr))
    {
        printf("Client: BindMoniker failed (%x)\n", hr);
        return;
    }

    // Try a couple QueryInterface calls into the object code, first a
    // QueryInterface to IPersistFile...

    hr = pIUnk->QueryInterface(IID_IPersistFile,
       (void **)&pIPersistFile);

    if (FAILED(hr)) {
        printf("Client: QueryInterface IPersistFile failed (%x)\n", hr);
        pIUnk->Release();
        return;
    }
```

```
    // Followed by a QueryInterface to ICustomInterface.
    hr = pIUnk->QueryInterface(IID_ICustomInterface,
                    (void **)&pICustomInterface);

    if (FAILED(hr)) {
        printf("Client: QueryInterface failed (%x)\n", hr);
        pIUnk->Release();
        pIPersistFile->Release();
        return;
    }

    // CustomReport() is the object function that displays the time and
    // date information on the object.
    hr = pICustomInterface->CustomReport();

    if (FAILED(hr))
    {
        printf("Client: pICustomInterface->CustomReport failed (%x)\n",
             hr);
        pIUnk->Release();
        pIPersistFile->Release();
        return;
    }

    // Clean up resources by calling release on each of the interfaces.
    pIPersistFile->Release();
    pICustomInterface->Release();
    pIUnk->Release();
    return;
}
```

URL Open Stream Functions

URL Open Stream (UOS) functions are ActiveX™ extensions to the Win32 API. They combine the familiarity of C-style programming with the power of COM. Yet using UOS functions requires knowledge of no more than two COM interfaces, **IStream** and **IBindStatusCallback**. UOS functions work equally well inside an ActiveX framework (for example, a component, a document or frame window, a subcomponent, or a scriptable object) or in a generic Internet context.

Every UOS function works in the same basic way: the caller implements an **IBindStatusCallback** interface (optional in some cases), then calls the function. The **URLOpenStream** and **URLOpenPullStream** functions require the caller to be on a thread that has a message loop (GetMessage/DispatchMessage). In the case of an ActiveX component, a message loop is a given if one of these functions is called from the main thread. For a stand-alone application without a user interface, a message loop is still necessary to use these functions.

With the UOS functions, you can:

- Download a URL to a file with a single function call. You can optionally get progress notifications in the background.
- Create a blocking-type stream (see Asynchronous Storage) with a single function that will block when you call IStream::Read. You can optionally get progress notifications in the background.
- Hook into the ActiveX client framework, if you like, simply by passing your this pointer.
- Configure callbacks using either the push or pull model (see **Asynchronous Monikers**).

URL open stream functions use services from URL Monikers and WinInet, providing all the caching and thread-synchronization features of those components. In addition, if your code is in an ActiveX container, the UOS functions handle all the host binding operations, automatically doing the right things to ensure an efficient and successful download. That is, these functions will determine whether your code is hosted within a container that supports the **IBindHost** interface and will use this interface if it is present. Otherwise, they will work without it.

APPENDIX A

Compatibility with OLE 1 and 16:32-Bit Interoperability

Note In moving from OLE 1 to OLE 2, the following changes in terminology were made:

- The OLE 1 term "server application" has been changed to "object application."
- The OLE 1 term "client application" has been changed to "container application."

Compatibility implies that an OLE 1 client application can contain OLE 2 embedded and linked objects and that an OLE 1 server application can create objects to be embedded in and linked to by OLE 2 containers. OLE provides these capabilities by means of a built-in compatibility layer in the core code, which includes a set of functions for conversion.

Interoperability is not the same as OLE 1 to OLE 2 compatibility. Interoperability implies that a 16-bit OLE application can interact with a 32-bit application running on the same system. Interoperability between 16- and 32-bit implementations means that 16-bit and 32-bit implementations can call each other.

Compatibility with OLE 1

Compatibility between OLE 2 and OLE 1 applications is achieved through the implementation of two types of special remoting objects, called stubs and proxies. The stub is instantiated on the object, or server, application's side of the process; the proxy is instantiated on the container, or client, application's side. These special stubs and proxies use DDE to communicate rather than LRPC. When an OLE 2 object makes a call to a function in an OLE 1 client application, for example, the stub intercepts the call and responds appropriately. For the most part, this response simulates the response that an OLE 2 object or container would make. However, in a few cases the behavior is different or special **HRESULT** (SCODE) values are returned.

This chapter discusses the issues that affect applications that must be compatible with an earlier or more recent version of OLE and describes the functions that promote this compatibility.

Working with OLE 1 Clients

This section describes some of the idiosyncrasies of working with OLE 1 clients.

A successful call to the **IOleClientSite::GetContainer** method returns a pointer to the container's **IOleContainer** interface. If the container does not support the **IOleContainer** interface, OLE_E_NOT_SUPPORTED is returned. All OLE 1 clients fall in this category, as do OLE 2 containers that do not support linking to their embedded objects.

The **IOleClientSite::ShowObject** method, a request to make the embedded or linked object visible, always returns OLE_E_NOT_SUPPORTED when called on an OLE 1 client**.** The purpose of this method is to help make the user model work smoothly; its failure does not effect OLE functionality.

When an OLE 1 client contains an OLE 2 object and the object is activated or the **OleUpdate** function is called, the aspect of the data returned will always be DVASPECT_CONTENT. This is because OLE 1 clients have no concept of a **FORMATETC** data structure. This situation may occur when an iconic OLE 2 object is pasted from an OLE 2 container into an OLE 1 container. When the object is first pasted, its presentation remains iconic. With the next update, however, the object's content picture is returned.

OLE 1 clients can link to OLE 2 objects only if the link source:

- is represented by a file moniker or a generic composite moniker consisting of a file moniker and one item moniker.
- is not an embedded OLE 2 object.

An OLE 1 client can contain an incompatible link when a linked object is pasted from an OLE 2 container into the OLE 1 client or when an OLE 2 container saves the data to an OLE 1 file, to allow the OLE 1 version of the application access to its data. When the OLE 1 client loads the incompatible link, the link is converted to an embedded object and assigned the class name "Ole2Link." The OLE 1 client cannot connect to the link source. However, if the newly embedded object is then pasted into an OLE 2 container using the Clipboard, or converted to an OLE 2 object using the **OleConvertOLESTREAMToIStorage** function, it will be converted back to its original state as an OLE 2 linked object.

When the link source for an OLE 1 linked object changes its name, the link can remain intact only if the file moniker for the link source has changed. That is, if the link source is a range of cells within an OLE 2 spreadsheet application and the name of the file that contains the cell range changes, OLE will track the link. However, if the name of the cell range changes, the link will break.

Pasting an OLE 2 linked object into an OLE 1 client document and then calling the **OleCopyFromLink** function to convert it to an embedded object will fail if the data transfer object provided by the link source does not support the **IPersistStorage** interface. Creating an embedded object always requires native data, and the **IPersistStorage** interface provides access to native data.

Working with OLE 1 Servers

This section describes some of the known idiosyncrasies of embedding or linking OLE 1 objects.

As with OLE 2 objects, either the **IPersistStorage::InitNew** method or the **IPersistStorage::Load** method must be called to properly initialize a newly instantiated OLE 1 object before any other OLE calls are made. The **InitNew** method should be called to initialize a newly created object; the **Load** method should be called for existing objects. If one of the **OleCreate** helper functions or the **OleLoad** function is being used, these functions make the **IPersistStorage** call, eliminating the need to make the call directly. When an OLE 2 container with an OLE 1 embedded or linked object calls the **IDataObject::GetData** method or the **IDataObject::GetDataHere** method, the container can anticipate support for a smaller set of formats and storage mediums than would be supported for an OLE 2 object. The following table lists the combinations that can be supported.

Tymed Formats	Data Formats
TYMED_MFPICT	CF_METAFILEPICT
TYMED_GDI	CF_BITMAP
TYMED_HGLOBAL	cfNative, CF_DIB, and other OLE 1 server formats

For the aspect value of DVASPECT_ICON, only TYMED_MFPICT with CF_METAFILEPICT is supported. The icon returned from the **IDataObject::GetData** or **IDataObject::GetDataHere** call will always be the first icon (index 0) in the executable object application.

Several methods typically called by containers have unique implementations for OLE 1. The **IPersistStorage::IsDirty** method is defined to return S_OK if the object has changed since its last save to persistent storage; S_FALSE if it has not changed. When an OLE 2 container with an OLE 1 embedded object calls the **IPersistStorage::IsDirty** method, the compatibility code always returns S_OK when the server is running, because there is no way to determine if the object has in fact changed until the File Close or File Update command is selected. S_FALSE is returned when the server is not running.

An OLE 2 implementation of **IOleObject::IsUpToDate** can return either S_OK if the object is up-to-date, S_FALSE if it is not up-to-date, or OLE_E_UNAVAILABLE if the object cannot determine whether it is up-to-date. The OLE 1 implementation always returns either E_NOT_RUNNING, if the object is in the loaded state, or S_FALSE, if the server is running.

The OLE 1 implementation of the **IOleItemContainer::EnumObjects** method always returns OLE_E_NOTSUPPORTED because it is not possible for an OLE 1 server to enumerate its objects.

The **IOleObject::Close** method takes a save option as a parameter that indicates whether the object should be saved before the close occurs. For OLE 2 objects, there are three possible save options: OLECLOSE_SAVEIFDIRTY, OLECLOSE_NOSAVE, and OLECLOSE_PROMPTSAVE. The OLE 1 implementation of the **IOleObject::Close** method treats OLECLOSE_PROMPTSAVE as equivalent to OLECLOSE_SAVEIFDIRTY, because it is not possible to require an OLE 1 server to prompt the user.

OLE 2 containers cannot expect an OLE 1 object to activate in-place; all OLE 1 objects support activation in a separate, open window.

OLE 1 servers do not support linking to their embedded objects. It is up to OLE 2 containers with OLE 1 embedded objects to prevent a possible link from occurring. Containers can call the **CoIsOle1Class** function to determine at Clipboard copy time if a data selection being copied is an OLE 1 object. If the **CoIsOle1Class** function returns TRUE, indicating that the selection is an OLE 1 object, the container should not offer the Link Source format. Link Source must be available for a linked object to be created.

OLE 2 containers can store multiple presentations for an OLE 1 object. However, only the first presentation format is sent to the container when the OLE 1 server closes. After that, the server is in the

process of closing down and cannot honor requests for any more formats. Therefore, only the first presentation cache will be updated. The rest will be out of date (perhaps blank) if the object has changed since the last update.

Because OLE 1 servers do not update the cache for every change to an embedded object until the user selects the File Update command, an OLE 2 container may not be obtaining the latest data from the server. By calling the **IOleObject::Update** method, the container can obtain the latest object data.

An OLE 1 embedded (not linked) object does not notify its container that its data has changed until the user chooses File Update or File Close. Therefore, if an OLE 2 container registers for a data-change notification on an OLE 2 object in a particular format, it should be aware that it will not be notified immediately when the data changes.

When an OLE 1 object is inserted into a container document and then closed without an update being invoked, the container document is not saved. Neither are the correct streams for the object written into storage. Any subsequent loading of the object by the container will fail. To protect against this, containers can keep data available after the object closes without updating by implementing the following:

```
OleCreate();              \\ to insert the object
OleRun();                 \\ if OLERENDER_NONE was specified
IOleObject::Update();     \\ to get snapshot of data
OleSave();
IOleObject::DoVerb();
```

OLE 1/Registry Compatibility Information

This section discusses the registry and how to handle compatibility issues between OLE 1 and related to OLE 2.

OLE 1 Compatibility Subkeys

To handle two-way compatibility, the OLE 2 compatibility layer creates OLE 2-style entries for OLE 1 classes it discovers and places them under the CLSID key. When installing an OLE 2 object application on a system that contains an OLE 1 version of the same application, it might be necessary to add "**AutoConvertTo**" or "**TreatAs**" subkeys to the *original* OLE 1 application portion of the registry. For more information, see "When the OLE 1 Version is Overwritten," later in this section.

Information for OLE 1 Applications Subkey Entries

To maintain compatibility with OLE 1, you must include specific OLE 1 information. The "**server**" key entry should contain a full path to the application. The entries for verbs must start with 0 as the primary verb and be consecutively numbered.

```
Hkey_Classes_Root\Ole2isvrotl\Protocol\Stdfileediting\Server =
    C:\Samp\Isvrotl.Exe
Hkey_Classes_Root\Ole2isvrotl\Protocol\Stdfileediting\Verb\0 = &Edit
Hkey_Classes_Root\Ole2isvrotl\Protocol\Stdfileediting\Verb\1 = &Open
```

OLE 1 Application Entries

When an OLE 1 class is inserted into an OLE 2 container for the first time, a new subkey, **CLSID**, is added *to the original OLE 1 registration information* by the OLE 2 compatibility layer. The value given to this key is a CLSID assigned by OLE 2 to this OLE 1 class as shown in the following portion of the registry.

<*OLE1ClassName*> = <*OLE1UserTypeName*>
    Protocol
        StdFileEditing
        Verb
    CLSID = <*CLSID*>

ProgID and OLE 1 Compatibility

Programmatic identifiers are not guaranteed to be unique so they can be used only where name collisions are manageable, such as in achieving compatibility with OLE 1. Also, the ProgID is the "class name" used for an OLE 2 class when it is placed in a server application (OLE 1 server).

Note Because OLE 2 provides a built-in OLE 1/OLE 2 compatibility layer, rarely will an OLE 2 class that is insertable in an OLE 2 container not be insertable in an OLE 1 container.

Version-dependent Identifiers

The <*VersionDependentProgID*> is the string to use when OLE 1 needs to contact OLE 2 using DDE. Version-dependent ProgID-to-CLSID conversions must be specific, well-defined, and one-to-one.

Inserting an OLE 2 Object in an OLE 1 Application

If a particular class is insertable in an OLE 1 container, the "ProgID" *root* key will contain a **Protocol\StdFileEditing** subkey with appropriate subkeys **Verb**, **Server**, and so on, as in OLE 1. The **Server** that should be registered here is the full path to the executable file of the OLE 2 object application. An OLE 1 container uses the path and executable file names to launch the OLE 2 object application. The initialization of this application, in turn, loads the OLE 2 compatibility layer. This layer handles subsequent interactions with the OLE 1 container (client), turning them into OLE 2-like requests to the OLE 2 application. An OLE 2 object application doesn't have to take any special action beyond setting up these registry entries to make objects insertable into an OLE 1 container.

The ProgID key and its subkeys appear in the registry as shown in the following example, where "***<ProgId>***" is the key, and "**Insertable**," "**Protocol**," "**StdFileEditing**," "**Verb**," and so on are subkeys.

```
<ProgId> = <MainUserTypeName>
    Insertable      // class is insertable in OLE 2 containers
    Protocol
        StdFileEditing      // OLE 1 compatibility info; present if, and only if,
                                    // objects of this class are insertable in OLE 1 containers.
            Server = <full path to the OLE 2 object application>
            Verb
                0 = <verb 0>      // Verb entries for the OLE 2 application must start with zero as the
                1 = <verb 1>      // primary verb and run consecutively.
    CLSID = <CLSID>      // The corresponding CLSID. Needed by GetClassFile.
    Shell      // Windows 3.1 File Manager Info
        Print
        Open
        Command = <appname.exe> %1
```

To summarize, any root key that has either an **Insertable** or a **Protocol\StdFileEditing** subkey is the ProgID (or OLE 1 class name) of a class that should appear in the Insert Object dialog box. The value of that root key is the name displayed in the Insert Object dialog box.

The values of each key in the example below are used for registering the "Ole 2 In-Place Server Outline" sample application. Set these values as required and used by your application.

Specifying Unregistered Verbs

It is possible to register OLE 1 object applications (servers) with no specified verbs. In such cases, the application has a single implied verb that is understood to be "edit" by OLE 1 containers. When an OLE 2 application calls the **IOleObject::EnumVerbs** method (or the **OleRegEnumVerbs** function) on an object of this class, one verb is enumerated. By default, the name of the verb is "Edit." To avoid having the string "Edit" as the enumerated verb, a key can be included in the registry. This key is

\Software\Microsoft\OLE1\UnregisteredVerb = <*verbname*>

(where *verbname* is the value that will be returned from the enumeration) and allows for localization of the default verb to a specified string to accommodate a specific language.

Note Do not attempt to register a verb (under ProgID\Protocol\StdFileEditing) for an application that did not register the verb itself.

Accommodating OLE 1 Versions of the Object Application

When the OLE 1 version of an object application is superseded by an OLE 2 version and the OLE 2 version is to be installed in the user's system, such as when upgrading, two situations can arise:

- An OLE 1 version of the application is present on the user's system and the installation process overwrites the OLE 1 executable with the OLE 2 version.
- An OLE 1 version of the application is present on the user's system and the user chooses not to overwrite it with the OLE 2 version.

Note Even if the OLE 1 object application is not on the user's system, the install/setup program for the OLE 2 object application should register the application as capable of servicing its OLE 1 objects. To do this, follow the guidelines presented under "When the OLE 1 Version is Overwritten," and add the following entry to the *CLSID root key*:

<**OLE 1 class name**>/**CLSID =** <*CLSID of OLE 1 application*>

When the OLE 1 Version is Overwritten

When the OLE 1 object application is replaced by the OLE 2 version, do the following. (For the purpose of illustration, the OLE 1 object application is referred to as Ole 1 In-Place Server Outline while the OLE 2 version is Ole 2 In-Place Server Outline, as shown in the dialog box illustrations that follow).

1. Register Ole 2 In-Place Server Outline.
2. Modify (with your install/setup program) the *original* registry entries of Ole 1 In-Place Server Outline by changing the executable path to point to the Ole 2 In-Place Server Outline executable.

   For example, the Server subkey for the OLE 1 executable, named **svrapp.exe** in this example, changes from

   OLE1ISvrOtl\Protocol\StdFileEditing\Server = svrapp.exe

   to

   OLE1ISvrOtl\Protocol\StdFileEditing\Server = isvrotl.exe

   where **isvrotl.exe** is the name of the OLE 2 object application.

   Next, proceed to either step 3 or step 4, depending on whether or not you want OLE 1 objects converted automatically to the OLE 2 format.
3. If Ole 1 In-Place Server Outline objects will be converted automatically to the Ole 2 In-Place Server Outline format when the application is saved, create or modify the following registration database entries:
   a. Modify the *original registry entry* of Ole 1 In-Place Server Outline by changing the "Value of the ProgID **=** Main User Type Name" key of the entry to Ole 2 In-Place Server Outline.

      For example, where **OLE1ISvrOtl** is the ProgID of the OLE 1 application,

      OLE1ISvrOtl = Ole 1 In-Place Server Outline

      becomes

      OLE1ISvrOtl = Ole 2 In-Place Server Outline
   b. Modify the *original registry entry* of Ole 1 In-Place Server Outline by adding a **NotInsertable** subkey under the ProgID key. For example:

      `HKEY_CLASSES_ROOT\OLE1SvrOtl\NotInsertable`

   Note Since the original registry entries for the OLE 1 server application remain (but with a pointer to the OLE 2 object application as shown in the preceding *step 3a*), the OLE 1 ProgID will appear in the Insert Object dialog box of any OLE 2 container application installed on the system. The **NotInsertable** subkey mentioned in *step 3b* prevents the ProgID of the OLE 1 application from appearing in the Insert Object dialog box of OLE 2 containers. The **NotInsertable** subkey overrides any **Insertable** subkey entries for that ProgID key.

   c. Set the "AutoConvertTo = CLSID" subkey entry for Ole 1 In-Place Server Outline *under the CLSID key* to the CLSID of Ole 2 In-Place Server Outline. (See also "OLE 1 Compatibility Subkeys").

      CLSID\{*CLSID of OLE 1 app.***}\AutoConvertTo = {***CLSID of OLE 2 app.***}**

   Note You can obtain the CLSID of the OLE 1 object application for inclusion in registration entry file by calling **CLSIDFromProgID**.

   d. Modify the *original registry entry* of Ole 1 In-Place Server Outline by setting the verbs to those of Ole 2 In-Place Server Outline.

      For example, change

      OLE1ISvrOtl\Protocol\StdFileEditing\Verb\0 = &Edit

      to

OLE1ISvrOtl\Protocol\StdFileEditing\Verb\0 = &Edit

OLE1ISvrOtl\Protocol\StdFileEditing\Verb\1 = &Open

4. If the user is allowed to open Acme Draw 1.0 objects and save them back to disk in the Acme Draw 1.0 format:
   a. Set the "TreatAs = CLSID" entry to the CLSID of Ole 2 In-Place Server Outline using the following form.

      CLSID\{*CLSID of OLE 1 app.***}\TreatAs = {***CLSID of OLE 2 app.***}**

   b. Set the Ole 1 In-Place Server Outline verbs to those of Ole 2 In-Place Server Outline as described in the preceding step 3.

When the OLE 1 Version is Not Overwritten

You can install your program so the OLE 1 object application (Ole 1 In-Place Server Outline) is *not* replaced by the OLE 2 version (Ole 2 In-Place Server Outline). Instead, the user is allowed to open Ole 1 In-Place Server Outline objects with the Ole 2 In-Place Server Outline and save them back to disk in the Ole 1 In-Place Server Outline format. To do this, set the "TreatAs = CLSID" entry (Ole 1 In-Place Server Outline's portion of the registry) to the CLSID of Ole 2 In-Place Server Outline (as in step 4 above).

If the OLE 1 version of the application is not overwritten, or if the user does not want to set the "Treat As" option, register the OLE 2 version as a separate and new application.

{ewc msdncd, EWGraphic, bsd23514 0 /a "SDK.WMF"}

Installing a new version of a server application.

Upgrading Applications

When an OLE 1 server is upgraded to an OLE 2 object application, several issues arise. A primary issue is whether the OLE 2 application will replace the OLE 1 application or both versions will coexist. If only the newer version will be available to the user, it is best to convert objects from the older version of the application automatically to the new version format. Objects can be converted on a global basis, where all objects of a specific class are converted, or on a more selective basis, where only some objects are converted. Conversion can be either automatic, under programmatic control, or under the control of a user.

The ability to detect whether an object is from an OLE 1 server is helpful for implementing conversion functionality. The OLE 2 implementation of the **IPersistStorage::Load** method can check for a stream named "\1Ole10Native." The "\1Ole10Native" stream contains a DWORD header whose value is the length of the native data that follows. The existence of this stream indicates that the data is coming from an OLE 1 server. Applications can check whether a storage object contains an object in an OLE 1 format by calling the **ReadFmtUserTypeStg** method and examining the contents of *pcfFormat.* This is where the OLE 1 class name would appear.

In the **IPersistStorage::Save** method, objects that are being permanently converted should be written back to storage in the new format and the "\1Ole10Native" stream should be deleted. The conversion bit in the storage should also be cleared once the conversion to the new format is complete.

To allow manual conversion of an old OLE 1 object to the new OLE 2 version, the OLE 2 object application must put the OLE 1 server's ProgID (OLE 1 server class name) into the registry under the **CLSID\{...}\Conversion\Readable\Main** entry. This entry indicates that the OLE 2 application can read its OLE 1 data format; the 'Clipboard format' of the OLE 1 data is the ProgID (that is, the class name) of the OLE 1 object.

To get a CLSID for an OLE 1 server, the **CLSIDFromProgId** function or the **CLSIDFromString** function must be called. That is, an OLE 1 application cannot be assigned a CLSID from an OLE 2 application with **uuidgen.exe**, **CoCreateGuid**, or by using a GUID from a range assigned by Microsoft. Because all OLE 1 CLSIDs are expected to fall in a specific range, OLE 1 CLSIDs are assigned with the **CLSIDFromProgId** function.

Refer to the appendix called "Registering Object Applications" for detailed information on the required registry entries for upgraded applications.

Functions to Support Compatibility

The following functions enable applications to determine whether an object class is from OLE 1, and to support conversion between OLE 1 and OLE 2 storage formats.

OLE 1 Compatibility Functions	Description
CoIsOle1Class	Determines if a given CLSID represents an OLE 1 object.
OleConvertIStorageToOLESTRAM	Converts the specified storage object from OLE 2 structured storage to the OLE 1 storage model.
OleConvertIStorageToOLESTREAMEx	Converts the specified storage object from OLE 2 structured storage to the OLE 1 storage model.
OleConvertOLESTREAMToIStorage	Converts the specified object from the OLE 1 storage model to an OLE 2 structured storage object.
OleConvertOLESTREAMToIStorageEx	Converts the specified object from the OLE 1 storage model to an OLE 2 structured storage object.

16:32 Bit Interoperability

The 32-bit implementation of OLE 2 is compatible with OLE 1 applications in the manner described above. Interoperability is achieved by converting calls made on objects in one model into calls suitable in another model. For example, if a call is made by a 16-bit application to an object that is a 32-bit implementation, the OLE 16:32 interoperability (thunk) layer takes care of marshalling and converting parameters between the two models.

Thunk Layer Operation

The thunk layer replaces the 16-bit implementations of OLE 2 with a new set of binaries that allow the 16-bit implementations to call the updated 32-bit implementation of OLE 2. These new binaries, collectively known as the thunk layer, forward all OLE 2 method and function calls to the 32-bit implementation of OLE. The OLE thunk layer inserts itself between implementations of different models. Whenever an interface is passed between implementation types, the system provides a thunk proxy and stub.

For example, a call to the **CreateFileMoniker** function returns a pointer to a moniker. When a 16-bit implementation calls the **CreateFileMoniker** function, the system thunks the call to the 32-bit OLE 2 implementation and creates a file moniker using a 32-bit implementation. When the call returns through the OLE thunk layer, the moniker pointer is translated into a 16/32 proxy.

A 16/32 proxy sets up an object in the 16-bit address space that, when called, converts each parameter for the specific method into the appropriate values for calling the 32-bit implementation. It then performs a model switch from 16-bit to 32-bit. The call is made on the 32-bit implementation, and then another model switch is made from the 32-bit to 16-bit implementation. In the process of switching back, any output parameters are converted into their 16-bit counterparts.

The system keeps track of the model of each pointer. For example, if a 16-bit application calls a 32-bit application, it passes a 16-bit pointer to an object. The thunk layer will create a 32/16 proxy to let the 32-bit application call the 16-bit application's object. In response, the 32-bit application calls back to the 16-bit application, passing the pointer it received. In this direction, the pointer being held by the 32-bit application holds is to a 32/16 proxy. The thunk layer also detects that the 32-bit pointer being passed in is a proxy, and passes the original pointer it started with to the 16-bit application. Thus, the 16-bit application sees the same pointer it passed in. This works when passing from 16 to 32 to 16, or from 32 to16 to 32. For more information about thunking, see the *Win32 SDK*.

Interoperability Using Standard Interfaces (OLE 2 Defined)

In its 32-bit implementation, OLE 2 is provided by the operating system and can translate OLE-defined interfaces and methods. Existing OLE 2 applications will continue to work as before. The following combinations of interoperability using the standard interfaces defined by OLE 2 are supported:

- 16 to 16 (in-process server)
- 16 to 16 (local server)
- 16 to 32 (local server)
- 32 to 32 (in-process server)
- 32 to 32 (local server)
- 32 to 16 (local server)

32- to16-bit (in-process server) interoperability using the standard interfaces is not supported.

Interoperability Using Custom Interfaces (either MIDL or Manually Written Marshalling)

Due to major architectural changes in the OLE communication layer, 16-bit applications that require remoting of custom interfaces are not supported at this time. However, 16-bit applications that use custom interfaces to an inprocess server should still function correctly. The following combinations of interoperability using custom interfaces are supported:

- 16 to 16 (inprocess server)
- 32 to 32 (inprocess server)
- 32 to 32 (local server using MIDL-generated stubs)

The following combinations of interoperability using custom interfaces are not supported:

- 16 to 16 (local server)
- 16 to 32 (all combinations)
- 32 to 16 (all combinations)

Programmers who have written applications that provide custom interfaces must rewrite the custom interface marshalling code to conform to the RPC-based communication layer. However, information about writing a 16-bit custom-marshalling library is not available at this time. For more information about RPC, see the *RPC Programmer's Guide and Reference*.

OLE Serialized Property Set Format

Persistent property sets provide a way to store information within file system entities. It is recommended that to create and manage them, you use the **IPropertySetStorage** and **IPropertyStorage** interfaces as described in the section of the Structured Storage chapter entitled **Persistent Property Sets**.

Property sets are made up of a tagged section of values, with the section uniquely identified by a Format Identifier (FMTID). Every property consists of a *property identifier* and a *type indicator* that represents a *value*. Each value stored in a property set has a unique property identifier that distinguishes the property. The type indicator describes the representation of the data in the value.

When you use the **IPropertySetStorage** and **IPropertyStorage** interfaces, you do not have to deal directly with the OLE serialized property set format structure. However, for those who are interested, this Appendix describes this format.

All data elements within a property set are stored in Intel representation (that is, in little-endian byte order).

OLE defines a standard, serialized data format for property sets. When you are dealing directly with the serialized format, and not with the interfaces, property sets have the following characteristics:

- Property sets allow for different applications to create their own independent property sets to serve the application's needs.
- Property sets can be stored in a single **IStream** instance or in an **IStorage** instance containing multiple streams. Indeed, in the abstract, property sets are simply another data type that can be stored in many different forms of an in-memory or on-disk storage. For recommended conventions on creating the string name for the storage object, see the section "Naming Property Sets" later in this appendix.
- Property sets allow for a dictionary of displayable names to be included to further describe the contents. A set of conventions for choosing property names is recommended. For more information on this optional dictionary, see "Property ID 0" later in this appendix.

The property set stream is divided into three major parts:

- Header
- FORMATID/offset pair
- Section containing the actual property set values

The overall length of the property set stream must be less than or equal to 256K. The following sections of this chapter describe the individual components that make up the property set data format, as shown in the previous figure.

> **Note** Previous versions of this document described extensions to the property set stream with more than one section allowed, but that has been revised now to provide for one section in the property stream. The one exception is the DocumentSummaryInformation property set, described in the section The DocumentSummaryInformation Property Set.

Property Set Header

At the beginning of the property set stream is a header. It consists of a byte-order indicator, a format version, the originating operating system version, the CLSID, and a reserved field.

The following pseudo-structure illustrates the header:

```
typedef struct tagPROPERTYSETHEADER
{
    // Header
    WORD   wByteOrder ;  // Always 0xFFFE
    WORD   wFormat ;     // Always 0
    DWORD   dwOSVer ;     // System version
    CLSID  clsID ;       // Application CLSID
    DWORD  reserved ;    // Should be 1
} PROPERTYSETHEADER;
```

Byte-Order Indicator

The byte-order indicator is a WORD and should always hold the value 0xFFFE. This is the same as the Unicode byte-order indicator. This value is always written in Intel byte order and, thus, appears in the file or stream as 0xFE, 0xFF.

Format Version

The Format Version is a WORD used to indicate the format version of this stream. It should always be zero. The format-version indicator should be checked when reading the property set. If it is not zero, then the stream was written to a different specification and cannot be read by code developed according to the OLE 2 specification.

Originating OS Version

This DWORD should hold the kind of operating system in the high-order word and the version number of the operating system in the low-order word. Possible values for the operating system are:

Operating System	Value
32-Bit Windows (Win32)	0x0002
Macintosh	0x0001
16-Bit Windows (Win16)	0x0000

For Windows, the operating system version is the low-order word returned by the **GetVersion** function. On Windows, the following code would correctly set the version of the originating operating system:

```
#ifdef WIN32
dwOSVer = (DWORD)MAKELONG( LOWORD(GetVersion()), 2 ) ;
#else
dwOSVer = (DWORD)MAKELONG( LOWORD(GetVersion()), 0 ) ;
#endif
```

CLSID

The CLSID is that of a class that can display and/or provide programmatic access to the property values. If there is no such class, it is recommended that you set this value the same as the Format identifier (see below). Alternately, you can set this value to all zeroes; however, using the Format identifier allows more flexibility in the future.

Reserved

This DWORD is reserved for future use. Writers of property sets should set this value to 1; readers of property sets should ensure that this value is at least 1. An exception to this guideline is that, for the DocumentSummaryInformation property set, this value may be 2.

Format Identifier/Offset Pair

The second part of the property set stream contains one Format Identifier (FMTID)/Offset Pair. The FMTID is the name of the property set; it uniquely identifies how to interpret the contents of the following section. The Offset is the distance of bytes from the *start of the whole stream* to where the section begins.

The following structure is helpful in dealing with Format Identifier/Offset Pairs:

```
typedef struct tagFORMATIDOFFSET
{
    FMTID  fmtid ;     // Name of the section
    DWORD  dwOffset ;  // Offset for the section
} FORMATIDOFFSET;
```

Format Identifiers

Property set values are stored in a section that is tagged with a unique format identifier. For example, the FMTID for the OLE Summary Information property set is:

```
F29F85E0-4FF9-1068-AB91-08002B27B3D9
```

To define a FMTID for the Summary Information property set, you would use the **DEFINE_GUID** macro in an include file for the code that manipulates the property set:

```
DEFINE_GUID(FMTID_SummaryInformation, 0xF29F85E0, 0x4FF9, 0x1068,
0xAB, 0x91, 0x08, 0x00, 0x2B, 0x27, 0xB3, 0xD9);
```

Anywhere in your code you need to use the FMTID for the Summary Information property set, you can access it through the FMTID_SummaryInformation variable.

When storing property sets in **IStorage** instances, you need to convert the FMTID to a string name for the storage object.

Allocating Format Identifiers

FMTIDs are created and represented in the same way as OLE CLSIDs and interface identifiers. To create a unique FMTID, use the UUIDGEN.EXE program included in the *Win32 SDK*.

Section

This is the third part of the property set stream, as shown in Figure C.2. A section contains:

- Byte count for the section (which is inclusive of the byte count itself)
- Array of 32-bit Property ID/Offset pairs
- Array of property Type Indicators/Value pairs

Offsets are the distance from the start of the section to the start of the property (type, value) pair. This allows a section to be copied as an array of bytes without any translation of internal structure.

The following pseudo-structures illustrate the format of a section:

```
typedef struct tagPROPERTYSECTIONHEADER
{
    DWORD  cbSection ;    // Size of Section
    DWORD  cProperties ;  // Count of Properties in section
} PROPERTYSECTIONHEADER;

typedef struct tagPROPERTYIDOFFSET
{
    DWORD  propid;    // Name of property
    DWORD  dwOffset;  // Offset from start of section to that property
} PROPERTYIDOFFSET;

typedef struct tag SERIALIZEDPROPERTYVALUE
{
    DWORD  dwType;   // Property Type
    BYTE   rgb[];    // Property Value
} SERIALIZEDPROPERTYVALUE ;
```

Size of Section

This DWORD indicates the size (in bytes) of the section. Because the section size is the first four bytes, you can copy sections as an array of bytes. The section size should always be a multiple of four.

For example, an empty section (one with zero properties in it) would have a byte count of eight (the DWORD byte count and the DWORD count of properties). The section itself would contain the eight bytes:

```
08 00 00 00 00 00 00 00
```

Count of Properties

This DWORD gives a count of the property values in the section. A property set may contain any number of property values. Readers must be able to handle the case where there are zero properties.

Property Identifiers/Offset Pairs

Following the Count of Properties is an array of Property Identifiers/Offset Pairs. Property identifiers are 32-bit values that uniquely identify a property within a section. The Offsets indicate the distance from the start of the section to the start of the property Type/Value Pair. Since the offsets are relative to the section, sections can be copied as an array of bytes.

Property identifiers are not sorted in any particular order. Properties can be omitted from the stored property set; readers must not rely on a specific order or range of property identifiers.

Type Indicators

After the table of Property Identifiers/Offset Pairs comes the actual properties. Each property is stored as a DWORD type, followed by the data value.

Type indicators and their associated values are described in the PROPVARIANT structure reference page.

All Type/Value pairs must begin on a 32-bit boundary. Thus, values may be followed with null bytes to align the subsequent pair on a 32-bit boundary. Given a count of bytes, the following code will calculate how many bytes are needed to align on a 32-bit boundary:

```
cbAdd = (((cbCurrent + 3) >> 2) << 2) - cbCurrent ;
```

Within a vector of values, each repetition of a simple scalar value smaller than 32 bits must align with its natural alignment rather than with a 32-bit alignment. In practice, this is only significant for types VT_UI1, VTUI2, VT_I2, and VT_BOOL (which have one- or two-byte natural alignment). All other types have four-byte natural alignment. Some types (VT_R8, etc.) actually have 8-byte natural alignment, but are stored as if they have 4-byte alignment. Therefore, a property value with type indicator VT_I2 | VT_VECTOR would be:

- A DWORD element count, followed by
- A sequence of packed two-byte integers with no padding between them.

Note that any 32-bit counts or property types that are stored as a part of a vector property element must also be 32-bit aligned.

A property value of type identifier VT_LPSTR | VT_VECTOR would be:

- A DWORD element count (**DWORD** *cElems*), followed by
- A sequence of strings (**char** *rgch*[]), each of which is preceded by a length count DWORD and may be followed by null padding to round to a 32-bit boundary.

Reserved Property Identifiers

As a designer of property sets you can use any Property identifier for your properties except 0, 1, and all values greater than or equal to 0x80000000. These Property identifier values are reserved for use by applications as follows.

Property ID 0

To enable users of property sets to attach meaning to properties beyond those provided by the type indicator, property ID 0 is reserved for an optional dictionary of displayable names for the property set.

The dictionary contains a count of entries in the list, followed by a list of dictionary entries.

```
typedef struct tagDICTIONARY
{
    DWORD  cEntries ;               // Count of entries in the list
    ENTRY  rgEntry[ cEntries ] ;    // Property ID/String pair
} DICTIONARY ;
```

Each dictionary entry in the list is a Property Identifier/String pair. This can be illustrated using the following pseudo-structure definition for a dictionary entry (it's a pseudo-structure because the sz[] member is variable in size):

```
typedef struct tagENTRY
{
    DWORD  propid ; // Property ID
    DWORD  cch ;       // Count of characters in the string
    char  sz[cch];     // Zero-terminated string
} ENTRY ;
```

Note the following about property set dictionaries:

- Property ID 0 does not have a type indicator. The DWORD that indicates the count of entries sits in the type indicator position.
- The count of characters in the string (*cch*) includes the zero character that terminates the string. When the codepage of the property set is not Unicode, this field is actually a *byte* count. This count may not exceed 256.
- The dictionary is entirely optional. Not all the names of properties in the set need appear in the dictionary. Conversely, not all names in the dictionary need to correspond to properties in the set. The dictionary should omit entries for properties assumed to be universally known by clients that manipulate the property set. Typically, names for the base property sets for widely accepted standards are omitted, but special purpose property sets may include dictionaries for use by browsers.
- Property names in the dictionary are stored in the code page indicated by Property ID 1 (see below). For ANSI code pages, each dictionary entry is byte-aligned. Thus, there is no padding between property names with Property ID 0. This is the only case where DWORD values (the property ID and property name length DWORDs) are not required to be aligned on 32-bit boundaries. For Unicode pages, each dictionary entry is 32-bit aligned.
- Property names that begin with the binary Unicode characters 0x0001 through 0x001F are reserved for future use.
- The property name associated with property ID 0 represents the name of the entire property set.

Sample Dictionary

The stock market data transfer example (see "Transferring Data Contained in Property Sets," in the Storage chapter of this Guide) might include a displayable name of "Stock Quote" for the entire set, and "Ticker Symbol" for PID_SYMBOL. If a property set contained just a symbol and the dictionary, the property set section would have a byte stream that looked like the following:

```
Offset     Bytes
; Start of section
0000    5C 01 00 00    ; DWORD size of section
0004    04 00 00 00    ; DWORD number of properties in section

; Start of PropID/Offset pairs
0008    01 00 00 00    ; DWORD Property ID (1 == code page)
000C    28 00 00 00    ; DWORD offset to property ID
0010    00 00 00 80    ; DWORD Property ID (0x80000000 == locale
                                                ID)
0014    30 00 00 00    ; DWORD offset to property ID
0018    00 00 00 00    ; DWORD Property ID (0 == dictionary)
001C    38 00 00 00    ; DWORD offset to property ID
0020    07 00 00 00    ; DWORD Property ID (3 == PID_SYMBOL)
0024    5C 01 00 00    ; DWORD offset to property ID

; Start of Property 1 (code page)
0028    01 00 00 00    ; DWORD type indicator (VT_12)
002C    B0 04          ; USHORT codepage (0x04b0 == 1200 ==
                                              unicode)
002E    00 00          ; Pad to 32-bit boundary

; Start of Property 0x80000000 (Local ID)
0030    13 00 00 00    ; DWORD type indicator (VT_U14)
0034    09 04 00 00    ; ULONG locale ID (0x0409 == American
                                             English)

; Start of Property 0 (the dictionary)
0038    08 00 00 00    ; DWORD number of entries in dictionary
                            (Note:  No type indicator)
003C    00 00 00 00    ; DWORD propid == 0 (the dictionary)
0040    0C 00 00 00    ; DWORD cch == wcslen(L"Stock Quote") +
                              sizeof(L'\0') == 12
0044    L"Stock Quote" ; wchar_t wsz(12)
005C    05 00 00 00    ; DWORD propid == 5 (PID_HIGH)
0060    0B 00 00 00    ; DWORD cch == wcslen(L"High Price") +
                              sizeof(L'\0') == 11
0064    L"High Price\0"; wchar_t wsz(11)
007A    00 00          ; padding for 32-bit alignment (necessary
                           because the codepage is unicode)
007C    07 00 00 00    ; DWORD propid == 7 (PID_SYMBOL)
0080    0E 00 00 00    ; DWORD cch - wcslen(L"Ticker Symbol\0")
                             == 14
0084    L"Ticker Symbol\0" ; wchar_t wsz(14)

//The dictionary would continue, but may not contain entries
   //for every possible property, and may contain entries for
   //properties that are not present. Also, entries need not be in
```

```
//order.
```

Property ID 1

Property ID 1 is reserved as an indicator of which code page (Windows) or Script (Macintosh) to use when interpreting the strings in the property set. All string values in the property set must be stored with the same code page. The originating operating system value in the property set header (PROPERTYSETHEADER::dwOSVer) determines whether the code page indicator corresponds to a Windows code page or Macintosh script.

When an application that is not the author of a property set changes a property of type string in the set, it should examine the code page indicator and take one of the following actions:

- Write the string in the format specified by the code page indicator.
- Replace and rewrite to change the code page.

If an application cannot understand this indicator, it should not modify the property. All creators of property sets must write a code page indicator; however, if the code page indicator is not present, the prevailing code page on the reader's machine must be assumed.

> **Note** If the **IPropertySetStorage** interface is used to create a property set, the code page indicator is automatically written.

Possible values for the code page are given in the Win32 API (see the **GetACP** function) and *Inside Macintosh Volume VI,* §*14-111*. For example, the code page US ANSI is represented by 0x04E4 (1252 in decimal) while the code page for Unicode is 0x04B0 (1200 in decimal).

It is recommended that the Unicode code page be used whenever possible, and use VT_LPWSTR instead of VT_LPSTR to avoid multibyte <-> Unicode conversions during storage and retrieval. Using the same code page for all property sets is the only way to achieve interoperable property sets on a worldwide basis. In either the Unicode or non-Unicode code page, note that the count at the start of a VT_LPSTR or VT_BSTR is a **byte** count and not a **character** count. This byte count includes the one or two zero bytes at the end of the string (the string's NULL terminator).

Property ID 1 is a VT_12 type and thus starts with a DWORD containing the value VT_12 followed by a USHORT indicating the code page.

Property ID 0x80000000

Property ID 0x80000000 (Locale Indicator) is reserved as an indication of the locale for which the property set is written. The default locale for a property set is the system's default locale (LOCALE_SYSTEM_DEFAULT). See the Win32 SDK for more information on LOCALE_SYSTEM_DEFAULT. The default is used in the event that the locale indicator does not exist in the property set.

Applications can choose to support the locale or just get the default behavior. It is recommended that applications allow users to specify a working locale. Such applications should write the user-specified locale identifier to the property. Applications that use the user's default locale (LOCALE_USER_DEFAULT) should write the user's default locale identifier to the property. See the Win32 SDK for more information on LOCALE_USER_DEFAULT.

> **Note** If the **IPropertySetStorage** interface is used to create a property set, the user's default locale is automatically written as the Locale Indicator.

Applications should also handle the case of a foreign object, which is one where the locale is not the application's locale, the user's locale, or the system's locale.

The locale indicator property is of type VT_U14, and therefore consists of a DWORD containing VT_U14, followed by a DWORD containing the Locale Identifier (LCID), as defined in the Win32 SDK.

Other Reserved Property Identifiers

Property identifiers with the high bit set (that is, negative values) are reserved for future use by Microsoft.

Storing Property Sets

Applications can expose some of the state of their documents so that other applications can locate and read that information. Some examples are a property set describing the author, title, and keywords of a document created with a word processor, or the list of fonts used in a document. This facility is not restricted to documents; it can also be used on embedded objects. Generally, access to property sets should be through the **IPropertySetStorage** and **IPropertyStorage** interfaces, but this section describes the previously recommended way.

> **Note** If you are storing a property set that is internal to your application, you might not want to follow the guidelines described below. If you want to expose your property set to other applications, you need to follow these guidelines.

To store a property set in a compound file:

1. Create an **IStorage** or **IStream** instance in the same level of the storage structure as its data streams. Prepend the name of your **IStorage** or **IStream** instance with "\005." Stream and storage names that begin with 0x05 are reserved for common property sets that can be shared among applications. Also, streams beginning with that value are limited to 256K. The names can be selected from either published names and formats, or by assigning the property set a FMTID and deriving the name from the FMTID according to the conventions described in the section "Naming Property Sets".
2. A property set may be stored in a single **IStream** instance or in an **IStorage** instance containing multiple streams and storages. In the case of an **IStorage** instance, the contained stream named "Contents" is the primary stream containing property values, where some values may be names of other streams or storage instances within the storage for this property set.
3. Specify the CLSID of the object class that can display and/or provide programmatic access to the property values. If there is no such class, the CLSID should be set to the property set's format identifier. For a property set that uses an **IStorage** instance, either set the CLSID of the **IStorage** instance to be the same as that stored in the Contents stream or to CLSID_NULL (the value in a newly created **IStorage** instance).
4. You have the option of specifying displayable names that form the contents of the dictionary.

Some applications can read only implementations of property sets stored as **IStream** instances. Applications should be written to expect that a property set may be stored in either an **IStorage** or **IStream** instance, unless the property set definition indicates otherwise. For example, the Summary Information property set's definition says that it can only be stored in a named **IStream** instance. In cases where you are searching for a property set and don't know whether it is a storage or stream, look for an **IStream** instance with your property set name first. If that fails, look for an **IStorage** instance.

To better understand storing property sets in an **IStorage** implementation, suppose there is a class of applications that edit information about animals. First, a CLSID (CLSID_AnimalApp) is defined for this set of applications, so they can indicate that they understand property sets containing animal information (FMTID_AnimalInfo), and others containing medical information (FMTID_MedicalInfo).

```
IStorage (File): "C:\OLE\REVO.DOC"
  IStorage: "\005AnimalInfo", CLSID = CLSID_AnimalApp
    IStream: "Contents"
      WORD wByteOrder, WORD wFmtVersion, DWORD dwOSVer,
      CLSID CLSID_AnimalApp, DWORD cSections...
      ...
      FMTID = FMTID_AnimalInfo
      Property: Type = PID_ANIMALTYPE, Type = VT_LPWSTR, Value = L"Dog"
      Property: Type = PID_ANIMALNAME, Type = VT_LPWSTR, Value = L"Revo"
      Property: Type = PID_MEDICALHISTORY, Type = VT_STREAMED_OBJECT,
```

```
            Value = "MedicalInfo"
    ...
  IStream: "MedicalInfo"
    WORD wByteOrder, WORD wFmtVersion, DWORD dwOSVer,
    CLSID CLSID_AnimalApp, DWORD cSections...
    ...
    FMTID = CLSID_MedicalInfo
    Property: Type = PID_VETNAME, Type = VT_LPWSTR, Value = L"Dr. Woof"
    Property: Type = PID_LASTEXAM, Type = VT_DATE, Value = ...
```

Note that the class identifiers of the **IStorage** interface and both property sets is CLSID_AnimalApp. This identifies any application that can display and/or provide programmatic access to these property sets. Any application can read the information within the property sets (the point behind property sets), but only applications identified with the class identifier of CLSID_AnimalApp can understand the meaning of the data in the property sets.

The DocumentSummaryInformation Property Set

The Microsoft Office Summary Information properties are stored in a separate stream from the standard Summary Information properties. The standard Summary Information property set is described in the section entitled "The Summary Information Property Set" under "Using Property Sets". The name of the stream that contains the Document Summary Information is:

```
"\005DocumentSummaryInformation"
```

The FMTID for the Microsoft Office Summary Information property set is:

```
D5CDD502-2E9C-101B-9397-08002B2CF9AE
```

Use the DEFINE_GUID macro to define the FMTID for the property set:

```
DEFINE_GUID(FMTID_DocumentSummaryInformation, 0xD5CDD502L, 0x2E9C,
0x101B, 0x93, 0x97, 0x08, 0x00, 0x2B, 0x2C, 0xF9, 0xAE);
```

This stream also has a separate section for the custom-defined properties. The format id for the section is

```
DEFINE_GUID(FMTID_UserDefinedProperties, 0xD5CDD505L, 0x2E9C, 0x101B,
0x93, 0x97, 0x08, 0x00, 0x2B, 0x2C, 0xF9, 0xAE);
```

The following table shows the added properties to the "DocumentSummaryInformation" stream for Office 95 applications.

Property Name	Property ID String	Property ID	VT Type
Category	PID_CATEGORY	0x00000002	VT_LPSTR
PresentationTarget	PID_PRESFORMAT	0x00000003	VT_LPSTR
Bytes	PID_BYTECOUNT	0x00000004	VT_I4
Lines	PID_LINECOUNT	0x00000005	VT_I4
Paragraphs	PID_PARCOUNT	0x00000006	VT_I4
Slides	PID_SLIDECOUNT	0x00000007	VT_I4
Notes	PID_NOTECOUNT	0x00000008	VT_I4
HiddenSlides	PID_HIDDENCOUNT	0x00000009	VT_I4
MMClips	PID_MMCLIPCOUNT	0x0000000A	VT_I4
ScaleCrop	PID_SCALE	0x0000000B	VT_BOOL
HeadingPairs	PID_HEADINGPAIR	0x0000000C	VT_VARIANT | VT_VECTOR
TitlesofParts	PID_DOCPARTS	0x0000000D	VT_LPSTR | VT_VECTOR
Manager	PID_MANAGER	0x0000000E	VT_LPSTR
Company	PID_COMPANY	0x0000000F	VT_LPSTR
LinksUpTo Date	PID_LINKSDIRTY	0x00000010	VT_BOOL

These properties have the following uses:

Category
: A text string typed by the user indicating what category the file belongs to (memo, proposal etc.). It is useful for finding files of same type.

PresentationTarget

Target format for presentation (35mm, printer, video etc.), from PowerPoint.

Bytes
: Number of bytes, from AFX.

Lines
: Number of lines, from AFX.

Paragraphs
: Number of paragraphs, from AFX.

Slides
: Number of slides, from PowerPoint

Notes
: Number of pages that contain notes, from PowerPoint

HiddenSlides
: Number of slides that are hidden, from PowerPoint

MMClips
: Number of sound or video clips, from PowerPoint

ScaleCrop
: Set to True (-1) when scaling of the thumbnail is desired. If not set, cropping is desired. FindFile 2.0 needs this.

HeadingPairs
: Internally used property indicating the grouping of different document parts and the number of items in each group. The titles of the document parts are stored in the PID_DOCPARTS property. The HeadingPairs property is stored as a vector of variants, in repeating pairs of VT_LPSTR and VT_I4 values. The VT_LPSTR value represents a heading name, and the VT_I4 value indicates the count of document parts under that heading. This property is used for providing the indentation for different groups on the 'sections' page.

TitlesofParts
: Names of document parts, from AFX. For Excel this is sheet names, for PowerPoint this is slide titles, for a binder this is document names, for Word it is the names of the documents in the master document.

Manager
: Manager of the project, from Project.

Company
: Company name, from Project.

LinksUpToDate
: Bool to indicate whether the custom links are dirty, for all applications.

Note As described in "12.3. Serialized Format for Property Sets" of the OLE 2.0 Design Specification, vector elements in the HeadingPairs and TitlesofParts properties should be aligned on 32 bit boundaries. However, in the DocumentSummaryInformation property set, when the code page of the property set is not Unicode, these elements must be packed.

C and C++ Design Considerations

Component objects contain data specific to the object and one or more interface implementations. The data is private and inaccessible from outside the object, while the interface implementations are public and you can access them through pointers.

Because interfaces are a binary standard, interface implementation is language independent. However, C++ is the preferred language because it supports many of the object-oriented concepts inherent in OLE. Using a procedural language such as C involves extra work, as summarized below:

- You must explicitly initialize VTBLs either at compile-time or run-time and you should not change them later. Once initialized with function pointers, those pointers should remain unchanged until the application shuts down. VTBLs in C++ are declared as constants to prevent them from being modified inadvertently. However, in C there is no way to ensure that a VTBL will remain unchanged.

  To overcome this difference, OLE provides a mechanism that lets C developers declare VTBLs as constants. To do so, place the following statement before the *#include* statement for the "ole2.h" header file:

  ```
  #define CONST_VTABLE
  ```

- Each method requires a pointer to the object that owns the method. In C++, all members are implicitly dereferenced off the *this* pointer. In C, there must be an additional first parameter passed to each method that is a pointer to the interface in which the method is declared.
- Methods in C++ can have identical names because methods are actually known by a name that is the result of concatenating the method name to the class name. Methods in C must have a unique name to designate the object with which they are associated.

For example, the following C++ code sample defines an implementation of **IUnknown::QueryInterface**. The method name is **QueryInterface** and there are two parameters: a REFIID and a pointer to where to return the requested interface instance.

```
CUnknown::QueryInterface (REFIID riid, LPVOID * ppvObj);
```

A similar C implementation would require a more complex name and an additional first parameter to indicate the object owning the method:

```
IUnknown_Doc_QueryInterface (LPUNKNOWN pUnk, REFIID riid,
    LPVOID * ppvObj);
```

The following sections demonstrate how to declare a component object in a few typical ways: using C nested data structures, C++ nested classes, and C++ multiple inheritance. The demonstration object, called *CObj*, derives from **IUnknown** and supports two interfaces that also derive from **IUnknown**, **InterfaceA** and **InterfaceB**. *CObj's* private data includes a pointer to another component object in the application (*m_pCDoc*), a count of all the external references to the object (*m_ObjRefCount*), and pointers to two interfaces implemented by other component objects and used by *CObj (m_pOleObj* and *m_pStg)*. All object members use the *m_* prefix to make it easy to distinguish between member variables and other variables.

Component Objects: C Nested Structures

C interface implementations comprise data structures nested within the object's data structure. Each interface structure contains a VTBL pointer as its first member (*pVtbl*), a pointer to the object (*pCObj*), and a count of the external references to the interface (*m_RefCount*). The order of the members in the interface structures is identical, to facilitate code sharing.

```
typedef struct CObj {
  ULONG            m_ObjRefCount;
  LPSTORAGE        m_pStg;
  LPOLEOBJECT      m_pOleObj;
  struct CDOC    *  m_pCDoc;

  struct InterfaceA  {
    LPVTBL        pVtbl;
    struct CObj  *  pCObj;
    ULONG        m_RefCount;
  } m_InterfaceA;

  struct InterfaceB {
    LPVTBL       pVtbl;
    struct Obj  *    pCObj;
    ULONG        m_RefCount;
  } m_InterfaceB;

} COBJ;
```

Component Objects: C++ Nested Classes

The next example shows how the same object is declared and initialized using the C++ nested class approach. As in the C example, the nested class declaration includes one data structure for each interface and four private data members: an object-level reference count, two interface pointers, and a pointer to the enclosing object. The private implementations of the **IUnknown** methods are called by the implementations declared for the derived interfaces. For each interface implementation, there is a structure containing a public constructor and destructor, private declarations of the interface methods, a private pointer to *CObj*, and an interface-level reference counter for debugging purposes. To allow the nested interface classes to access the private members of the outer class, each interface class is made a friend of the outer class.

The benefits of implementing with C++ nested classes lie in the ability to include initialization code and method implementation inline. However, inline declaration is for the convenience of illustration and is not required.

```
class  CObj  {

private:
  ULONG          m_ObjRefCount;
  LPSTORAGE      m_pStg;
  LPOLEOBJECT    m_pOleObj;
  CDOC    *      m_pCDoc;

public:
  CObj();
  ~CObj();

  struct CUnknown : IUnknown
{
  private:
  ULONG          m_RefCount;
  CObj  *        m_pCObj;

  public:
  CUnknown(CObj (pCObj)
  { m_pCObj = pCObj; m_RefCount = 0; }
  HRESULT QueryInterface(REFIID riid, LPVOID  * ppvObj)
  ULONG AddRef(void) { return ++m_ObjRefCount; }
  ULONG Release(void);
}
friend CUnknown;
CUnknown m_Unknown;

struct InterfaceA : InterfaceA
{
  private:
  ULONG          m_RefCount;
  CObj  *        m_pCObj;

  public:
  CInterfaceA(CObj *pCObj)
  { m_pCObj = pCObj; m_RefCount = 0; }
  HRESULT QueryInterface(REFIID riid, LPVOID  * ppvObj)
```

```
  ULONG AddRef(void) { return ++m_ObjRefCount; }
  ULONG Release(void);
  HRESULT MethodA1(LPVOID  * ppvObj);
  HRESULT MethodA2(DWORD dwArg);
}
friend CInterfaceA;
CInterfaceA m_InterfaceA;

struct InterfaceB : InterfaceB
{
  private:
  ULONG          m_RefCount;
  CObj  *        m_pCObj;

  public:
  CInterfaceB(CObj *pCObj)
  { m_pCObj = pCObj; m_RefCount = 0; }
  HRESULT QueryInterface(REFIID riid, LPVOID  * ppvObj)
  ULONG AddRef(void) { return ++m_ObjRefCount; }
  ULONG Release(void);
  HRESULT MethodB1(void);
  HRESULT MethodB2(DWORD dwArg1, DWORD dwArg2);
}
friend CInterfaceB;
CInterfaceB m_InterfaceB;
```

Multiple Inheritance

The next example illustrates the use of C++ multiple inheritance. There are two disadvantages to using multiple inheritance with OLE. First, it is not possible to have an interface-level reference count. For more information about reference counting, see Chapter 2, "The Component Object Model." Second, there is the potential for confusion over the interpretation of the class statement. A standard C++ multiple inheritance declaration implies the "is a" relationship where an object inherits implementations. In OLE, however, interfaces are attributes of the object and implementations are not inherited.

The main advantage to using multiple inheritance lies in its simplicity. Only the prototypes for each of the interface methods are listed; no interface data structures or class definitions are necessary.

Because both InterfaceA and InterfaceB inherit from **IUnknown**, it is not necessary to list **IUnknown** explicitly in the class statement. A single implementation of the **IUnknown** methods (**QueryInterface**, **AddRef**, and **Release**) is sufficient.

```
class  CObj : public InterfaceA, public InterfaceB
{
private:
  ULONG          m_ObjRefCount;
  LPSTORAGE      m_pStg;
  LPOLEOBJECT    m_pOleObj;
  CDOC  *        m_pCDoc;

public:
  CObj();
  ~CObj();

  HRESULT QueryInterface(REFIID riid, LPVOID  * ppvObj)
  ULONG AddRef(void) { return ++m_ObjRefCount; }
  ULONG Release(void);

  HRESULT MethodA1(LPVOID  * ppvObj);
  HRESULT MethodA2(DWORD dwArg);

  HRESULT MethodB1(void);
  HRESULT MethodB2(DWORD dwArg1, DWORD dwArg2);
};
```

Converting Mapping Modes

In Windows, mapping modes define how numbers relating to object sizes are to be passed and interpreted. The 32-bit version of OLE uses only one mapping mode, MM_HIMETRIC. The other Win32 mapping modes include MM_HIENGLISH, MM_LOENGLISH, MM_LOMETRIC, TWIPS, and PIXEL Modes (MM_ISOTROPIC and MM_ANISOTROPIC being PIXEL modes). When working with the visual presentation of OLE data, it is important to be aware of the mapping mode that OLE uses and how this mapping mode affects an application.

The mapping modes communicate physical sizes. For example, if an application using MM_HIMETRIC is to display a line ten centimeters long, the number of units would be 10,000. However, the line drawn on the screen would be ten centimeters long, regardless of the size of the video display area. The printed output would also be a line ten centimeters long.

> **Note** For those applications that use a mapping mode other than MM_HIMETRIC, the sample user interface library provides some functions that can be used to convert objects to and from MM_HIMETRIC units.

Because people read display screens from a greater distance than they do printed copy, most applications written for Windows display text in a larger size than they print it, using what is commonly referred to as *logical resolution*. For example, a ten-point font is easy enough to read on the printed page, but generally appears too small on a screen for comfortable reading. To afford more comfortable viewing, applications typically expand the size of the displayed text to some logical size. Using this approach, a column of text that is physically six inches wide might be eight inches wide on the screen, yet still print as a six-inch column.

While this display-enlargement scheme works well from the user's point of view, a problem can occur when pasting objects into container documents. It is possible to lose the correct size ratio between the pasted object and the text owned and displayed by the container. The result is that the container's text is scaled up for readability but the pasted object might not be. Consequently, applications must preserve the relative size and position of text and objects, meaning that if text uses logical resolution, it should scale objects accordingly.

In the examples shown in the figure below, a chart object has been pasted from a source application that uses physical size into documents of two different containers that use logical resolution for display of the text. The container displaying the object on the left has scaled up the chart object to the logical size of the adjacent text to maintain the object/text size ratio. That is, it has been enlarged from its physical size by an amount that maintains its proportion to the text of the container document. In the document on the right, the application displays the chart object at its physical size, with no scaling to logical resolution for display. Both documents print with the correct object-to-text-size ratio.

{ewc msdncd, EWGraphic, bsd23518 0 /a "SDK.WMF"}

In OLE, the units for specifying the size of drawn objects is MM_HIMETRIC, which means object sizes are in physical units. However, containers need not use the MM_HIMETRIC mapping mode to draw pasted objects to the display. Rather, they should should map objects to the screen in the same manner as text. That is, if the container application displays text to the screen using a mapping mode that enlarges it, objects should be mapped to the display in the same manner. Using the same mapping mode for both the text and objects is required to establish the correct object-to-text ratio as shown in the document on the left side of the preceding figure. Because most Windows applications use logical resolution for this type of display mapping, we suggest that OLE containers also use logical resolution and set up their mapping modes and coordinate transforms accordingly. This allows objects to be moved from one application to another without changing their displayed size.

ActiveX Control and Control Container Guidelines

The purpose of this appendix is to provide guidelines for implementing ActiveX controls and containers that will interoperate well with other controls and containers. This appendix defines the minimum set of interfaces, methods, and features that are required of ActiveX Controls and Containers to accomplish seamless and useful interoperability.

Overview of Control and Control Container Guidelines

An ActiveX Control is essentially a simple OLE object that supports the **IUnknown** interface. It will usually support a lot more interfaces in order to offer functionality, but all additional interfaces may be viewed as optional and as such, a control container should not rely on any additional interfaces being supported. By not specifying additional interfaces that a control must support, a control may efficiently target a particular area of functionality without having to support particular interfaces to qualify as a control. As always with OLE, whether in a control or a control container, it should never be assumed that an interface is available and standard return-checking conventions should always be followed. It is important for a control or control container to degrade gracefully and offer alternative functionality if an interface required is not available.

An ActiveX control container must be able to host a minimal ActiveX Control as specified in this appendix, it will also support a number of additional interfaces as specified in the Containers section of this appendix. There are a number of interfaces and methods that a container may optionally support, which are grouped into functional areas known as Component Categories. A container may support any combination of component categories, for example, a component category exists for **Databinding** and a container may or may not support the databinding functionality, depending on the market needs of the container. If a control needs databinding support from a container to function, then it will enter this requirement in the registry. This allows a control container to only offer for insertion those controls that it knows it can successfully host. It is important to note that Component Categories are specified as part of OLE and are not specific to ActiveX Controls, the controls architecture uses Component Categories to identify areas of functionality that an OLE component may support. Component categories are not cumulative or exclusive, so a control container can support one category without necessarily supporting another.

It is important for controls that require optional features, or features specific to a certain container to be clearly packaged and marketed with those requirements. Similarly containers that offer certain features or component categories must be marketed and packaged as offering those levels of support when hosting ActiveX controls. It is recommended that controls target and test with as many containers as possible and degrade gracefully to offer less or alternative functionality if interfaces or methods are not available. In a situation where a control cannot perform its designated job function without the support of a component category, then that category should be entered as a requirement in the registry in order to prevent the control being inserted in an inappropriate container.

These guidelines define those interfaces and methods that a control may expect a control container to support, although as always a control should check the return values when using **QueryInterface** or other methods to obtain pointers to these interfaces. A container should not expect a control to support anything more than the **IUnknown** interface, and these guidelines identify what interfaces a control may support and what the presence of a particular interface means.

Why are the ActiveX Control and Control Container Guidelines Important?

ActiveX Controls have become the primary architecture for developing programmable software components for use in a variety of different containers ranging from software development tools to end-user productivity tools. In order for a control to operate well in a variety of containers, the control must be able to assume some minimum level of functionality that it can rely on in all containers.

By following these guidelines, control and container developers make their controls and containers more reliable and interoperable, and ultimately, better and more usable components for building component-based solutions.

This appendix provides guidelines towards good interoperability. It is expected that new interfaces and component categories will develop over time, future versions of this appendix reflecting these changes will be made readily available through Microsoft. It is important to note that this appendix does not cover detailed semantics of the OLE interfaces; this is covered by the Win32 SDK documentation.

What to do When an Interface You Need is Not Available

This section states some fundamental rules that apply to all OLE programming. OLE programs should use **QueryInterface** to acquire interface pointers, and must check the return value. OLE applications cannot safely assume that **QueryInterface** will succeed, this requirement applies to all OLE applications. If the requested interface is not available (i.e., **QueryInterface** returns E_NOINTERFACE), the control or container must degrade gracefully, even if that means that it cannot perform its designated job function.

What's New in the Control and Control Container Guidelines?

This release of the guidelines embraces the concept of Component Categories which are a part of the OLE specification. In previous versions of this document component categories were loosely referred to as function groups and were used to identify areas of functionality that a container may optionally support, for this version there has been a definition of how component categories work for ActiveX Controls and some fundamental categories are identified. The use of component categories allows the relaxing of some of the previous rules that identified interfaces as being mandatory, and allows greater flexibility for controls to efficiently target certain areas of functionality without having to provide superfluous additional support in order to qualify as a control. This edition of the guidelines also discusses what the presence or absence of an interface means and what to do in that situation.

The remainder of this appendix is divided into four sections. The first discusses guidelines for implementing controls, the second discusses guidelines for implementing control containers, the third discusses component categories, and the fourth discusses general guidelines, relevant to both control and control container developers.

Controls

An ActiveX control is really just another term for OLE object or more specifically, COM object. In other words, a control, at the very least, is some COM object that supports the **IUnknown** interface and is also self-registering. Through **IUnknown::QueryInterface** a container can manage the lifetime of the control as well as dynamically discover the full extent of a control's functionality based on the available interfaces. This allows a control to implement as little functionality as it needs to, instead of supporting a large number of interfaces that actually don't do anything. In short, this minimal requirement for nothing more than **IUnknown** allows any control to be as lightweight as it can.

In short, other than **IUnknown** and self-registration, there are no other requirements for a control. There are however conventions that should be followed about what the support of an interface means in terms of functionality provided to the container by the control. This section then describes what it means for a control to actually support an interface, as well as methods, properties, and events that a control should provide as a baseline if it has occasion to support methods, properties, and events.

Self Registration for Controls

ActiveX controls must support self-registration by implementing the **DllRegisterServer** and **DllUnregisterServer** functions. ActiveX controls must register all of the standard registry entries for embeddable objects and automation servers.

ActiveX Controls must use the component categories API to register themselves as a control and register the component categories that they require a host to support and any categories that the control implements, see the Component Categories section of this appendix. In addition an ActiveX Control may wish to register the **control** keyword in order to allow older control containers such as VB4 to host them.

ActiveX Controls should also register the **ToolBoxBitmap32** registry key, although this is not mandatory.

The **Insertable** component category should only be registered if the control is suitable for use as a compound document object. It is important to note that a compound document object must support certain interfaces beyond the minimum **IUnknown** required for an ActiveX Control. Although an ActiveX Control may qualify as a compound document object, the control's documentation should clearly state what behavior to expect under these circumstances.

What Support for an Interface Means

Besides the **IUnknown** interface, an ActiveX Control—or COM Object for that matter—expresses whatever optional functionality it supports through additional interfaces. This is to say that no other interfaces are required above **IUnknown**. To that end, the following table lists the interfaces that an ActiveX Control might support, and what it means to support that interface. Further details about the methods of these interfaces are given in a later section.

Interface	Comments/What it Means to Support the Interface
IOleObject	If the control requires communication with its client site for anything other than events (see **IConnectionPointContainer**), then **IOleObject** is a necessity. When implementing this interface, the control must also support the semantics of the following methods: **SetHostNames**, **Close**, **EnumVerbs**, **Update**, **IsUpToDate**, **GetUserClassID**, **GetUserType**, **GetMiscStatus**, and the **Advise**, **Unadvise**, and **EnumAdvise** methods that work in conjunction with a container's **IAdviseSink** implementation. A control implementing **IOleObject** must be able to handle **IAdviseSink** if the container provides one; a container may not, in which case a control ensures, of course, that it does not attempt to call a non-existent sink.
IOleInPlaceObject	Expresses the control's ability to be in-place activated and possibly UI activated. This interface means that the control has a user interface of some kind that can be activated, and **IOleInPlaceActiveObject** is supported as well. Required methods are **GetWindow**, **InPlaceActivate**, **UIDeactivate**, **SetObjectRects**, and **ReactivateAndUndo**. Support for this interface requires support for **IOleObject**.
IOleInPlaceActiveObject	An in-place capable object that supports **IOleInPlaceObject** must also provide this interface as well, though the control itself doesn't necessarily implement the interface directly.
IOleControl	Expresses the control's ability and desire to deal with (a) mnemonics (**GetControlInfo**, **OnMnemonic** methods), (b) ambient properties (**OnAmbientPropertyChange**), and/or (c) events that the control requires the container to handle (**FreezeEvents**). Note that mnemonics are different than accelerators that are handled through **IOleInPlaceActiveObject**: mnemonics have associated UI and are active even when the control is not UI active. A control's support for mnemonics means that the control also knows how to use the container's **IOleControlSite::OnControlInfoChanged** method. Because this requires the control to know the container's site, support for mnemonics also

	means support for **IOleObject**. In addition, knowledge of mnemonics requires in-place support and thus **IOleInPlaceObject**. If a control uses any container-ambient properties, then it must also implement this interface to receive change notifications, as following the semantics of changes is required. Because ambient properties are only available through the container site's **IDispatch**, ambient property support means that the control supports **IOleObject** (to get the site) as well as being able to generate **IDispatch::Invoke** calls. The **FreezeEvents** method is necessary for controls that must know when a container is not going to handle an event—this is the only way for control to know this condition. If **FreezeEvents** is only necessary in isolation, such that other **IOleControl** methods are not implemented, then **IOleControl** can stand alone without **IOleObject** or **IOleInPlaceObject**.
IDataObject	Indicates that the control can supply at least (a) graphical renderings of the control (CF_METAFILEPICT is the minimum if the control has any visuals at all) and/or (b) property sets, if the control has any properties to provide. The methods **GetData**, **QueryGetData**, **EnumFormatEtc**, **DAdvise**, **DUnadvise**, and **EnumDAdvise** are required. Support for graphical formats other than CF_METAFILEPICT is optional.
IViewObject2	Indicates that the control has some interesting visuals when it is not in-place active. If implemented, a control must support the methods **Draw**, **GetAdvise**, **SetAdvise**, and **GetExtent**.
IDispatch	Indicates that the control has either (a) custom methods, or (b) custom properties that are both available via late-binding through **IDispatch::Invoke**. This also requires that the control provides type information through other **IDispatch** methods. A control may support multiple **IDispatch** implementations where only one is associated with IID_IDispatch—the others must have their own unique dispinterface identifiers. A control is encouraged to supply dual interfaces for custom method and property access, but this is optional if methods and properties exist.
IConnectionPointContainer	Indicates that a control supports at least one outgoing interface, such as events or property change notifications. All methods of this interface must be implemented if this interface is available at all, including **EnumConnectionPoints** which requires a separate object with **IEnumConnectionPoints**. Support for **IConnectionPointContainer** means that the object also supports one or more related

	objects with **IConnectionPoint** that are available through **IConnectionPointContainer** methods. Each connection point object itself must implement the full **IConnectionPoint** interface, including **EnumConnections**, which requires another enumerator object with the **IEnumConnections** interface.
IProvideClassInfo[2]	Indicates that the object can provide its own coclass type information directly through **IProvideClassInfo::GetClassInfo**. If the control supports the later variation **IProvideClassInfo2**, then it also indicates its ability to provide its primary source IID through **IProvideClassInfo2::GetGUID**. All methods of this interface must be implemented.
ISpecifyPropertyPages	Indicates that the control has property pages that it can display such that a container can coordinate this control's property pages with other control's pages when property pages are to be shown for a multi-control selection. All methods of this interface must be implemented when support exists.
IPerPropertyBrowsing	Indicates the control's ability to (a) provide a display string for a property, (b) provide pre-defined strings and values for its properties and/or (c) map a property dispID to a specific property page. Support for this interface means that support for properties through IDispatch as above is provided. If (c) is supported, then it also means that the object's property pages mapped through **IPerPropertyBrowsing::MapPropertyToPage** themselves implement **IPropertyPage2** as opposed to the basic **IPropertyPage** interface.
IPersistStream	See Persistence Interfaces section.
IPersistStreamInit	See Persistence Interfaces section.
IPersistMemory	See Persistence Interfaces section.
IPersistStorage	See Persistence Interfaces section.
IPersistMoniker	See Persistence Interfaces section.
IPersistPropertyBag	See Persistence Interfaces section.
IOleCache[2]	Indicates support for container caching of control visuals. A control generally obtains caching support itself through the OLE function **CreateDataCache**. Only controls with meaningful static content should choose to do this (although it is not required). If a control supports caching at all, it should simply aggregate the data cache and expose both **IOleCache** and **IOleCache2** interfaces from the data cache. A control implementing **IOleObject** must be able to handle **IAdviseSink** if the container provides one; a container may not, in which case a control ensures, of course, that it does not attempt to call a non-existent sink.
IExternalConnection	Indicates that the control supports external links to

	itself; that is, the control is not marked with OLEMISC_CANTLINKINSIDE and supports **IOleObject::SetMoniker** and **IOleObject::GetMoniker**. A container will never query for this interface itself nor call it directly as calls are generated from inside OLE's remoting layer.
IRunnableObject	Indicates that the control differentiates being loaded from being running, as some in-process objects do.

Persistence Interfaces

Objects that have a persistent state of any kind must implement at least one **IPersist*** interface, and preferably multiple interfaces, in order to provide the container with the most flexible choice of how it wishes to save a control's state.

If a control has any persistent state whatsoever, it must, as a minimum, implement either **IPersistStream** or **IPersistStreamInit** (the two are mutually exclusive and shouldn't be implemented together for the most part). The latter is used when a control wishes to know when it is created new as opposed to reloaded from an existing persistent state (**IPersistStream** does not have the created new capability). The existence of either interface indicates that the control can save and load its persistent state into a stream, that is, an instance of **IStream**.

Beyond these two stream-based interfaces, the **IPersist*** interfaces listed in the following table can be optionally provided in order to support persistence to locations other than an expandable **IStream**.

A set of component categories is identified to cover the support for persistency interfaces see the Component Categories section of this appendix.

Interface	Usage
IPersistMemory	The object can save and load its state into a fixed-length sequential byte array (in memory).
IPersistStorage	The object can save and load its state into an **IStorage** instance. Controls that wish to be marked **Insertable** as other compound document objects (for insertion into non-control aware containers) must support this interface.
IPersistPropertyBag	The object can save and load its state as individual properties written to **IPropertyBag** which the container implements. This is used for Save As Text functionality in some containers.
IPersistMoniker	The object can save and load its state to a location named by a moniker. The control calls **IMoniker::BindToStorage** to retrieve the storage interface it requires, such as **IStorage**, **IStream**, **ILockBytes**, **IDataObject**, etc.

While support for **IPersistPropertyBag** is optional, it is strongly recommended as an optimization for containers with Save As Text features, such as Visual Basic.

With the exception of **IPersistStream[Init]::GetSizeMax** and **IPersistMemory::GetSizeMax**, all methods of each interface must be fully implemented.

Optional Methods in Control Interfaces

Implementing an interface doesn't necessarily mean implementing all methods of that interface to do anything more than return E_NOTIMPL or S_OK as appropriate. The following table identifies the methods of the interfaces listed in the What Support for an Interface Means section that a control may implement in this manner. Check with the SDK OLE Reference documentation for full syntax and valid return values from these methods. Any method not listed here must be fully implemented if the interface is supported.

Method	Comments
IOleControl	
GetControlInfo, OnMnemonic	Mandatory for controls with mnemonics.
OnAmbientPropertyChange	Mandatory for controls that use ambient properties.
FreezeEvents	See Event Freezing in the General Guidelines section.
IOleObject	
SetMoniker	Mandatory if the control is not marked with OLEMISC_CANTLINKINSIDE
GetMoniker	Mandatory if the control is not marked with OLEMISC_CANTLINKINSIDE
InitFromData	Optional
GetClipboardData	Optional
SetExtent	Mandatory only for DVASPECT_CONTENT
GetExtent	Mandatory only for DVASPECT_CONTENT
SetColorScheme	Optional
DoVerb	See Note 1.
IOleInPlaceObject	
ContextSensitiveHelp	Optional
ReactivateAndUndo	Optional
IOleInPlaceActiveObject	
ContextSensitiveHelp	Optional
IViewObject2	
Freeze	Optional
Unfreeze	Optional
GetColorSet	Optional
IPersistStream[Init], IPersistMemory	
GetSizeMax	See Note 2.

1. A control with property pages must support **IOleObject::DoVerbs** for the

OLEIVERB_PROPERTIES and OLEIVERB_PRIMARY verbs. A control that can be active must support **IOleObject::DoVerbs** for the OLEIVERB_INPLACEACTIVATE verb. A control that can be UI active must also support **IOleObject::DoVerbs** for the OLEIVERB_UIACTIVATE verb.

2. If a control supports **IPersistStream[Init]** and can return an accurate value, then it should do so.

Class Factory Options

An ActiveX Control, by virtue of being a COM object, must have associated server code that supports control creation through **IClassFactory** as a minimum.

It is optional, not required, that this class object also supports **IClassFactory2** for licensing management. Only those vendors that are concerned about licensing need to support COM's licensing mechanism. In other words, because **IClassFactory2** is the only way to achieve COM-level licensing, this interface is required on the class object for those controls that wish to be licensed.

Properties

Although most controls do have properties, controls are not required to expose any properties and thus the control does not require **IDispatch**. If the control does have properties, there are no requirements for which properties a control must expose.

Methods (via IDispatch and Other dispinterfaces)

Although most controls do expose and support several methods, controls are not required to expose or support any methods and thus the control does not require **IDispatch**. If the control does have any methods, there are no requirements for which methods a control must expose.

Events in Controls

Although most controls do expose and fire several events, controls are not required to expose or fire any events and thus the control does not require **IConnectionPointContainer**. If the control does have any events, there are no requirements for which events a control must expose.

Property Pages

Support for property pages and per-property browsing is strongly recommended, but not required. If a control does implement property pages, then those pages should conform to one of the standard sizes: 250x62 or 250x110 dialog units (DLUs).

Ambient Properties for Controls

If a control supports any ambient properties at all, it must at least respect the values of the following ambient properties under the conditions stated in the following table using the standard dispids.

Ambient Property	Dispid	Comment/Conditions for Use
LocaleID	-705	If Locale is significant to the control, e.g. for text output
UserMode	-709	If the control behaves differently in user (design) mode and run mode
UIDead	-710	If the control reacts to UI events, then it should honor this ambient property
ShowGrabHandles	-711	If the control support in-place resizing of itself
ShowHatching	-712	If the control support in-place activation and UI activation
DisplayAsDefault	-713	Only if the control is marked OLEMISC_ACTSLIKEBUTTON (which means that support for keyboard mnemonics is provided, thus **IOleControl::GetControlInfo** and **IOleControl::OnMnemonic** must be implemented).

As described previously, use of ambients requires both **IOleControl** (for **OnAmbientPropertyChange** as a minimum) as well as **IOleObject** (for **SetClientSite** and **GetClientSite**).

The OLEMISC_SETCLIENTSITEFIRST bit may not necessarily be supported by a container. In these circumstances, a control must resort to default values for the ambient properties that it requires.

Using the Container's Functionality

The previous sections have described some of the necessary caller-side support that an ActiveX Control must have in order to access certain features of its container. The following table describes a control's usage of container-side interfaces and when such usage would occur.

Interface	Container Object	Usage
IOleClientSite	Site	Controls that implement **IOleObject** call **IOleClientSite** methods as part of the standard OLE embedding protocol, specifically the methods **SaveObject**, **ShowObject**, **OnShowWindow** (only if a separate-window activation state is supported), **RequestNewObjectLayout**, and **GetContainer** (if communication with other controls is desired). The **GetMoniker** method is only used when the control can be linked to externally, that is, the control is not marked with OLEMISC_CANTLINKINSIDE.
IOleInPlaceSite	Site	Controls that have an in-place activate and possibly a UI active state will call **IOleInPlaceSite** methods (generally all of them with the exception of **ContextSensitiveHelp**) as part of the standard OLE in-place activation protocol.
IAdviseSink	Site	Control calls **OnDataChange** if the control supports **IDataObject**, **OnViewChange** if the control supports **IViewObject2**, and **OnClose**, **OnSave**, and **OnRename** if the control supports **IOleObject**.
IOleControlSite	Site	If supported, control calls **OnControlInfoChanged** when mnemonics change, **LockInPlaceActive** and **TransformCoords** if events are fired (the latter method is only used if coordinates are passed as event arguments), **OnFocus** and **TranslateAccelerator** if the

		control has a UI active state, and **GetExtendedControl** if the control wants to look at extended-control (container-owned) properties.
IDispatch (ambient properties)	Site	Used to access ambient properties.
IPropertyNotifySink	Varies	A control must generate **OnChanged** and **OnRequestEdit** for any control properties that are marked as **[bindable]** and **[request]**, respectively.
Other event sink interfaces	Varies	A control that has outgoing interfaces other than **IPropertyNotifySink** will be handed other interface pointers of the correct IID to the control's **IConnectionPoint::Advise** implementations (which are usually found in sub-objects of the control). A control always knows how to call its own event interfaces since the control defines those interfaces.
IOleInPlaceFrame	Frame	Used when a control has an in-place UI active state that requires frame-level tools or menu items.
IOleInPlaceUIWindow	Document	Used only when a control has an in-place UI active state that requires document-level or pane-level UI tools. This is rare.

Containers

An ActiveX Control container is an OLE container that supports the following additional features:

1. Embedded objects from in-process servers
2. In Place activation
3. OLEMISC_ACTIVATEWHENVISIBLE
4. Event Handling

ActiveX Control Containers must provide support for all of these features.

The following sections describe the specific interfaces, methods, and other features that are required of ActiveX Control Containers. Required Interfaces, Optional Methods, Misc. Status Bits Support, Keyboard Handling, Storage Interfaces, Ambient Properties, Extended Properties, Events, Methods, Message Reflection, and Automatic Clipping. The last section describes how to gracefully degrade when a particular control interface is not supported.

Required Interfaces

The table below lists the ActiveX Control Container interfaces, and denotes which interfaces are optional, and which are mandatory and must be implemented by control containers.

Interface	Required?	Comments
IOleClientSite	Yes	
IAdviseSink	No	Only when the container desires (a) data change notifications (controls with **IDataObject**), (b) view change notification (controls that are not active and have **IViewObject[2]**), and (c) other notifications from controls acting as standard embedded objects.
IOleInPlaceSite	Yes	
IOleControlSite	Yes	
IOleInPlaceFrame	Yes	
IOleContainer	Yes	See Note 1.
IDispatch for ambient properties	Yes	See Note 2 and Ambient Properties for Controls section
Control Event Sets	Yes	See Note 2.
ISimpleFrameSite	No	**ISimpleFrameSite** and support for nested simple frames is optional.
IPropertyNotifySink	No	Only needed for containers that (a) have their own property editing UI which would require updating whenever a control changed a property itself or (b) want to control **[requestedit]** property changes and other such data-binding features.
IErrorInfo	Yes	Mandatory if container supports dual interfaces. See Note 2.
IClassFactory2	No	Support is strongly recommended.

1. **IOleContainer** is implemented on the document or form object (or appropriate analog) that holds the container sites. Controls use **IOleContainer** to navigate to other controls in the same document or form.
2. Support for dual interfaces is not mandatory, but is strongly recommended. Writing ActiveX Control Containers to take advantage of dual interfaces will afford better performance with controls that offer dual interface support.

ActiveX Control containers must support OLE Automation exceptions. If a control container supports dual interfaces, then it must capture automation exceptions through **IErrorInfo**.

Optional Methods

An OLE component can implement an interface without implementing all the semantics of every method in the interface, instead returning E_NOTIMPL or S_OK as appropriate. The following table describes those methods that an ActiveX Control container is not required to implement (i.e. the control container can return E_NOTIMPL).

The table below describes optional methods; note that the method must still exist, but can simply return E_NOTIMPL instead of implementing real semantics. Note that any method from a mandatory interface that is not listed below must be considered mandatory and may not return E_NOTIMPL.

Method	Comments
IOleClientSite	
SaveObject	Necessary for persistence to be successfully supported.
GetMoniker	Necessary only if the container supports linking to controls within its own form or document.
IOleInPlaceSite	
ContextSensitiveHelp	Optional
Scroll	May return S_FALSE with no action.
DiscardUndoState	Can return S_OK with no action.
DeactivateAndUndo	Deactivation is mandatory; Undo is optional.
IOleControlSite	
GetExtendedControl	Necessary for containers that support extended controls.
ShowPropertyFrame	Necessary for containers that wish to include their own property pages to handle extended control properties in addition to those provided by a control.
TranslateAccelerator	May return S_FALSE with no action.
LockInPlaceActive	Optional
IDispatch (Ambient properties)	
GetTypeInfoCount	Necessary for containers that support non-standard ambient properties.
GetTypeInfo	Necessary for containers that support non-standard ambient properties.
GetIDsOfNames	Necessary for containers that support non-standard ambient properties.
IDispatch (Event sink)	
GetTypeInfoCount	The control knows its own type information, so it has no need to call this.
GetTypeInfo	The control knows its own type

	information, so it has no need to call this.
GetIDsOfNames	The control knows its own type information, so it has no need to call this.
IOleInPlaceFrame	
ContextSensitiveHelp	
GetBorder	Necessary for containers with toolbar UI (which is optional)
RequestBorderSpace	Necessary for containers with toolbar UI (which is optional)
SetBorderSpace	Necessary for containers with toolbar UI (which is optional)
InsertMenus	Necessary for containers with menu UI (which is optional)
SetMenu	Necessary for containers with menu UI (which is optional)
RemoveMenus	Necessary for containers with menu UI (which is optional)
SetStatusText	Necessary only for containers that have a status line
EnableModeless	Optional
TranslateAccelerator	Optional
IOleContainer	
ParseDisplayName	Only if linking to controls or other embeddings in the container is supported, as this is necessary for moniker binding.
LockContainer	As for **ParseDisplayName**
EnumObjects	Returns all ActiveX Controls through an enumerator with **IEnumUnknown**, but not necessarily all objects (since there's no guarantee that all objects are ActiveX Controls; some may be regular Windows controls).

Miscellaneous Status Bits Support

ActiveX Control Containers must recognize and support the following OLEMISC_ status bits:

Status Bit	Required?	Comments
ACTIVATEWHENVISIBLE	Yes	
IGNOREACTIVATEWHENVISIBLE	No	Needed for inactive and windowless control support. See Note 1.
INSIDEOUT	No	Not generally used with ActiveX Controls but rather with compound document embeddings. Note this is contrary to some SDK documentation that says this bit must be set for the ACTIVATEWHENVISIBLE bit to be set.
INVISIBLEATRUNTIME	Yes	Designates a control that should be visible at design time, but invisible at run time.
ALWAYSRUN	Yes	
ACTSLIKEBUTTON	No	Support is normally mandatory although it is not necessary for document style containers.
ACTSLIKELABEL	No	Support is normally mandatory although it is not necessary for document style containers.
NOUIACTIVATE	Yes	
ALIGNABLE	No	
SIMPLEFRAME	No	See Simple Frame Site Containment section.
SETCLIENTSITEFIRST	No	Support for this bit is recommended but not mandatory.
IMEMODE	No	

1. The IGNOREACTIVATEWHENVISIBLE bit is for containers hosting inactive and windowless controls. The IGNOREACTIVATEWHENVISIBLE bit is introduced as part of the ActiveX Controls 96 specification, see this documentation for more details.

Keyboard Handling in Controls

Keyboard handling support for the following functionality is strongly recommended, although it is recognized that it is not applicable to all containers.

- Support for OLEMISC_ACTSLIKELABEL and OLEMISC_ACTSLIKEBUTTON status bits.
- Implementing the **DisplayAsDefault** ambient property (if it exists, it can return FALSE).
- Implementing tab handling, including tab order.

Some containers will use ActiveX Controls in traditional compound document scenarios. For example, a spreadsheet may allow a user to embed an ActiveX Control into a worksheet. In such scenarios, the container would do keyboard handling differently, because the keyboard interface should remain consistent with the user's expectations of a spreadsheet. Documentation for such products should inform users of differences in control handling in these different scenarios. Other containers should endeavor to honor the above functionality correctly, including Mnemonic handling.

Storage Interfaces

Control containers must be able to support controls that implement **IPersistStorage**, **IPersistStream**, or **IPersistStreamInit**. Optionally, a container can support any other persistence interfaces such as **IPersistMemory**, **IPersistPropertyBag**, and **IPersistMoniker** for those controls that provide support.

Once an ActiveX Control Container has chosen and initialized a storage interface to use (**IPersistStorage**, **IPersistStream**, **IPersistStreamInit**, etc), that storage interface will remain the primary storage interface for the lifetime of the control, i.e. the control will remain in possession of the storage. This does not preclude the container from saving to other storage interfaces.

ActiveX Control Containers do not need to support a save as text mechanism, thus using **IPersistPropertyBag** and the associated container-side interface **IPropertyBag** are optional.

Ambient Properties

At a minimum, ActiveX Control containers must support the following ambient properties using the standard dispids.

Ambient Property	Dispid	Comments/Conditions
LocaleID	-705	
UserMode	-709	For containers that have different user and run environments.
DisplayAsDefault	-713	For those containers where a default button is relevant.

Extended Properties, Events and Methods

ActiveX Control Containers are not required to support extended controls. However, if the control container does support extended properties, then it must support the following minimal set:

- Visible
- Parent
- Default
- Cancel

Currently, extended properties, events, and methods do not have standard dispids.

Message Reflection

It is strongly recommended that an ActiveX Control container supports message reflection. This will result in more efficient operation for subclassed controls. If message reflection is supported, the **MessageReflect** ambient property must be supported and have a value of TRUE. If a container does not implement message reflection, then the OLE CDK creates two windows for every sub-classed control, to provide message reflection on behalf on the control container.

Automatic Clipping

It is strongly recommended that an ActiveX Control container supports automatic clipping of its controls. This will result in more efficient operation for most controls. If automatic clipping is supported, the **AutoClip** ambient property must be supported and have a value of TRUE.

Automatic clipping is the ability of a container to ensure that a control's drawn output goes only to the container's current clipping region. In a container that supports automatic clipping, a control can paint without regard to its clipping region, because the container will automatically clip any painting that occurs outside the control's area. If a container does not support automatic clipping, then CDK-generated controls will create an extra parent window if a non-null clipping region is passed.

Degrading Gracefully in the Absence of an Interface

Because a control may not support any interface other than **IUnknown**, a container has to degrade gracefully when it encounters the absence of any particular interface.

One might question the usefulness of a control with nothing more than **IUnknown**. But consider the advantages that a control receives from a container's visual programming environment (such as VB) when the container recognizes the object as a control:

1. A button for the object appears in a toolbox.
2. One can create an object by dragging it from the toolbox onto a form.
3. One can give the object a name that is recognized in the visual programming environment.
4. The same name in (3) above can be used immediately in writing any other code for controls on the same form (or even a different form).
5. The container can automatically provide code entry points for any events available from that object.
6. The container provides its own property browsing UI for any available properties.

When an object isn't recognized as a control, then it potentially loses all of these very powerful and beneficial integration features. For example, in Visual Basic 4.0 it is very difficult to really integrate some random object that is not a control in the complete sense, but may still have properties and events. Because VB 4's idea of a control is very restrictive the object does not gain any of the integration features above. But even a control with **IUnknown**, where the mere lifetime of the control determines the existence of some resource, should be able to gain the integration capabilities described above.

As current tools require a large set of control interfaces to gain any advantage, controls are generally led to over-implementation, such that they contain more code than they really need. Controls that could be 7K might end up being 25K, which is a big performance problem in areas such as the Internet. This has also led to the perception that one can only implement a control with one tool like the CDK because of the complexity of implementing *all* the interfaces—and this has implications when a large DLL like OC30.DLL is required for such a control, increasing the working set. If not all interfaces are required, then this opens up many developers to writing very small and light controls with straight OLE or with other tools as well, minimizing the overhead for each control.

This is why this appendix recognizes a control as any object with a CLSID and an **IUnknown** interface. Even with nothing more than IUnknown, a container with a programming environment should be able to provide at least features #3 and #4 from the list above. If the object provides a ToolBoxBitmap32 registry entry, it gains #1 and #2. If the object supplies **IConnectionPointContainer** (and **IProvideClassInfo** generally) for some event set, it gains #5, and if it supports **IDispatch** for properties and methods, it gains #6, as well as better code integration in the container.

In short, an object should be able to implement as little as **IDispatch** and one event set exposed through **IConnectionPointContainer** to gain all of those visual features above.

With this in mind, the following table describes what a container might do in the absence of any possible interface. Note that only those interfaces are listed that the container will directly obtain through **QueryInterface**. Other interfaces, like **IOleInPlaceActiveObject**, are obtained through other means.

Interface	Meaning of Interface Absence
IViewObject2	The control has no visuals that it will draw itself, so has no definite extents to provide. In run-time, the container simply doesn't attempt to draw anything when this interface is absent. In design time, the container must at least draw some kind of default

	rectangle with a name in it for such a control, so a user in a visual programming environment can select the object and check out its properties, methods, and events that exist. Handling the absence of **IViewObject2** is critical for good visual programming support.
IOleObject	The control doesn't need the site whatsoever, nor does it take part in any embedded object layout negotiation. Any information (like control extents) that a container might expect from this interface should be filled in with container-provided defaults.
IOleInPlaceObject	The control doesn't go in-place active (like a label) and thus never attempts to activate in this manner. Its only activation may be its property pages.
IOleControl	Control has no mnemonics and no use of ambient properties, and doesn't care if the container ignores events. In the absence of this interface, the container just doesn't call its methods.
IDataObject	The control provides no property sets nor any visual renderings that could be cached, so the container would choose to cache some default presentation in the absence of this interface (support for CF_METAFILEPICT, specifically) and disable any property-set related functionality.
IDispatch	The control has no custom properties or methods. The container does not need to try to show any control properties in this case, and should disallow any custom method calls that the container doesn't recognize as belonging to its own extended controls (that may support methods and properties). As extended controls generally delegate certain **IDispatch** calls to the control, an extended control should not expect the control to have **IDispatch** at all.
IConnectionPointContainer	The control has no events, so the container doesn't have to think about handling any.
IProvideClassInfo[2]	The control either doesn't have type information or events, or the container needs to go into the control's type information through the control's

	registry entries. The existence of this interface is an optimization.
ISpecifyPropertyPages	The control has no property pages, so if the container has any UI that would invoke them, the container should disable that UI.
IPerPropertyBrowsing	The control has no display name itself, no predetermined strings and values, and no property to page mapping. This interface is nearly always used for generating container user interface, so such UI elements would be disabled in the absence of this interface.
IPersist*	The control has no persistent state to speak of, so the container doesn't have to worry about saving any control-specific data. The container will, of course, save its own information about the control in its own form or document, but the control itself has nothing to contribute to that information.
IOleCache[2]	The object doesn't support caching. A container can still support caching by just creating a data cache itself using **CreateDataCache**.

Component Categories

OLE's component categories allow a software component's abilities and requirements to be identified by entries in the registry. In a scenario where a container may not wish to or not be able to support an area of functionality, such as databinding for example, the container will not wish to host controls that require databinding in order to perform their job function. Component Categories allow areas of functionality such as databinding to be identified, so that the control container can avoid those controls that state it to be a requirement. Component Categories are specified separately as part of OLE and are not specific to the ActiveX Control architecture, the specification for component categories includes a set of APIs for manipulation of the component category registry keys.

What are Component Categories and how do they work?

Component Categories identify those areas of functionality that a software component supports and requires, a registry entry is used for each category or identified area of functionality. Each component category is identified by a globally unique identifier (GUID), when a control is installed it registers itself as a control in the system registry using the component category ID for control, see the Self Registration for Controls section. Within the control's self registration it will also register those component categories that it implements and those component categories that it requires a container to support in order to successfully host the control.

When a control container is offering controls to the user to insert, it only allows the user to select and instantiate those controls that will be able to function adequately in that environment. For example, if the control container does not support databinding, then the container will not allow the user to select and instantiate those controls that have an entry in the registry signifying that they require the databinding component category. A common dialog for control insertion and APIs to handle the registry entries are available.

Component categories are not cumulative or exclusive, a control can require any mix of component categories to function. A control that has no required entries for component categories may be expected to be capable of functioning in any control container and not require any specific functionality of a control container to function.

The following component categories are identified here, where necessary more detailed specifications of the categories may be available.

- **ISimpleFrameSite** control containment.
- Simple Databinding through the **IPropertyNotifySink** interface.
- Advanced Databinding (as supported by the additional databinding interfaces of VB4.0).
- Visual Basic private interfaces - **IVBFormat**, **IVBGetControl**
- Internet aware controls.
- Windowless controls.

This is not a definitive list of categories; further categories are likely to be defined in the future as new requirements are identified. An up-to-date list of component categories is available from Microsoft on their world wide web site, this list reflects those component categories that have been identified by Microsoft and any others that about which vendors have informed Microsoft.

It is important to remember that controls should attempt to work in as many environments as possible. If it is possible, the control should degrade its functionality when placed in a container that does not support certain interfaces. The purpose of component categories is to prevent a situation where the control is placed in an environment that is unsuitable and the control can not achieve its desired task. Generally, a control should degrade gracefully when interfaces are not present, a control may choose to advise the user with a message box that some functionality is not available or clearly document the functionality required of a control container for optimal performance.

Note older controls and containers do not make use of Component Categories and instead rely on the **control** keyword being present against the control in the registry. In order to be recognized by older containers controls may wish to register the **control** keyword in the registry, control developers should check that the control can successfully be hosted in such containers before doing this. Containers that use component categories may successfully use them to host older controls as the components category DLL handles the mapping, a separate category exists for older controls CATID_ControlV1 so that a container may optionally exclude them if necessary.

As Component Categories are identified by GUIDs it is possible for containers that offer particular specific functionality to have their own category IDs, generated using a GUID generation tool. However this can

possibly undermine the advantage of interoperability of controls and containers so it is preferred that wherever possible existing component categories be used. Vendors are encouraged to consult together when defining new component categories to ensure that they meet the common requirements of the marketplace, and follow the spirit of interoperability of ActiveX Controls.

Simple Frame Site Containment

A container control is an ActiveX Control that is capable of containing other controls. A group box that contains a collection of radio buttons is an example of a container control. Container controls should set the OLEMISC_SIMPLEFRAME status bit, and should call its container's ISimpleFrameSite implementation. An ActiveX Control container that supports Container Controls must implement **ISimpleFrameSite**.

CATID - {157083E0-2368-11cf-87B9-00AA006C8166} CATID_SimpleFrameControl

Simple Data Binding

The ActiveX Controls Architecture defines a data-binding mechanism, whereby an ActiveX Control can specify that one or more of its properties are bindable. In most cases, a data-bound control should not absolutely require data binding, so that it could be inserted into a container that does not support data binding. Obviously, in such a situation, the functionality of the control may be reduced.

CATID - {157083E1-2368-11cf-87B9-00AA006C8166} CATID_PropertyNotifyControl

Advanced Data Binding

There is a set of advanced data binding interfaces that allow a more complex databinding scenario to be supported. This component category covers that area of functionality.

CATID - {157083E2-2368-11cf-87B9-00AA006C8166} CATID_VBDataBound

Visual Basic private interfaces

Two interfaces that are implemented by Visual Basic are identified here for component categories. It is not expected that controls should require these categories as it is possible for controls to offer alternative functionality when these are not available.

The **IVBFormat** interface allows controls to better integrate into the Visual Basic environment when formatting data.

CATID - {02496840-3AC4-11cf-87B9-00AA006C8166} CATID_VBFormat

The **IVBGetControl** interface allows a control to enumerate other controls on the VB form.

CATID - {02496841-3AC4-11cf-87B9-00AA006C8166} CATID_VBGetControl

Internet-Aware Objects

There are certain categories identified to cover the persistency interfaces, these have been identified as a result of defining how controls function across the internet. A container that does not support the full range of persistency interfaces should ensure that it does not host a control that requires a combination of interfaces that it does not support. Details of the features required for internet aware controls are available in the ActiveX SDK.

The following tables describe the meaning for various categories as both implemented and required categories.

Required Categories	**Description**
CATID_PersistsToMoniker, CATID_PersistsToStreamInit, CATID_PersisitsToStream, CATID_PersistsToStorage, CATID_PersistsToMemory, CATID_PersistsToFile, CATID_PersistsToPropertyBag	Each of these categories are mutually exclusive and are only used when an object supports only one persistence mechanism at all (hence the mutual exclusion). Containers that do not support the persistence mechanism described by one of these categories should prevent themselves from creating any objects of classes so marked.
CATID_RequiresDataPathHost	The object requires the ability to save data to one or more paths and requires container involvement, therefore requiring container support for **IBindHost**.

Implemented Categories	**Description**
CATID_PersistsToMoniker, CATID_PersistsToStreamInit, CATID_PersistsToStream, CATID_PersistsToStorage, CATID_PersistsToMemory, CATID_PersistsToFile, CATID_PersistsToPropertyBag	Object supports the corresponding **IPersist*** mechanism for the category.

The following table provides the exact CATIDs assigned to each category:

Category	**CATID**
CATID_RequiresDataPathHost	0de86a50-2baa-11cf-a229-00aa003d7352
CATID_PersistsToMoniker	0de86a51-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStorage	0de86a52-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStreamInit	0de86a53-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStream	0de86a54-2baa-11cf-a229-00aa003d7352
CATID_PersistsToMemory	0de86a55-2baa-11cf-a229-00aa003d7352

CATID_PersistsToFile	0de86a56-2baa-11cf-a229-00aa003d7352
CATID_PersistsToPropertyBag	0de86a57-2baa-11cf-a229-00aa003d7352

Windowless Controls

The ActiveX Controls 96 specification includes a definition for windowless controls. Such controls do not operate in their own window and require a container to offer a shared window into which the control may draw, see the ActiveX SDK. Windowless controls are designed to be compatible with older control containers by creating their own window in that situation, windowless control containers should host windowed controls in the traditional way with no problem. It may however be useful for a container to distinguish those controls that can operate in a windowless mode, so an appropriate component category is defined.

CATID - {1D06B600-3AE3-11cf-87B9-00AA006C8166} CATID_WindowlessObject

General Guidelines

This section describes various features, hints and tips for ActiveX Control and ActiveX Control container developers to help ensure good interoperability between controls and control containers.

Overloading IPropertyNotifySink

Many ActiveX Control Containers implement a modeless property browsing window. If a control's properties are altered through the control's property pages, then the control's properties can get out of sync with the container's view of those properties (the control is always right, of course). To ensure that it always has the current values for a control's properties, an ActiveX Control Container can overload the **IPropertyNotifySink** interface (data binding) and also use it to be notified that a control property has changed. This technique is optional, and is not required of ActiveX Control Containers or ActiveX Controls.

Note that a control should use **IPropertyNotifySink::OnRequestEdit** only for data binding; it is free to use **OnChanged** for either or both purposes.

Container-Specific Private Interfaces

Some containers provide container-specific private interfaces for additional functionality or improved performance. Controls that rely on those container-specific interfaces should, if possible, work without those container-specific interfaces present so that the control functions in different containers. For example, Visual Basic® implements private interfaces that provide string formatting functionality to controls. If a control makes use of VB's private formatting interfaces, it should be able to run with default formatting support if these interfaces are not available. If the control can function without the private interfaces, it should take appropriate action (such as warn the user of reduced functionality) but continue to work. If this is not an option, then a component category should be registered as required to ensure that only containers supporting this functionality can host these controls.

Multi-Threaded Issues

Starting with Microsoft® Windows® 95 and Microsoft Windows NT® 3.51, OLE provides support for multi-threaded applications, allowing applications to make OLE calls from multiple threads. This multi-threaded support is called the apartment model, it is important that all OLE components using multiple threads follow this model. The apartment model requires that interface pointers are marshaled (using **CoMarshalInterface**, and **CoUnmarshalInterface**) when passed between threads. For more information about apartment model threading, refer to the Win32 SDK documentation, and the OLEAPT sample (in Win32® SDK).

Event Freezing

A container can notify a control that it is not ready to respond to events by calling **IOleControl::FreezeEvents(TRUE)**. It can un-freeze the events by calling **IOleControl::FreezeEvents(FALSE)**. When a container freezes events, it is freezing event processing, not event receiving; that is, a container can still receive events while events are frozen. If a container receives an event notification while its events are frozen, the container should ignore the event. No other action is appropriate.

A control should take note of a container's call to **IOleControl::FreezeEvents(TRUE)** if it is important to the control that an event is not missed. While a container's event processing is frozen, a control should implement one of the following techniques:

1. Fire the events in the full knowledge that the container will take no action.
2. Discard all events that the control would have fired.
3. Queue up all pending events and fire them after the container has called **IOleControl::FreezeEvents(FALSE)**.
4. Queue up only relevant or important events and fire them after the container has called **IOleControl::FreezeEvents(FALSE)**.

Each technique is acceptable and appropriate in different circumstances. The control developer is responsible for determining and implementing the appropriate technique for the control's functionality.

Container Controls

As described above, container controls are ActiveX Controls that visually contain other controls. The ActiveX Controls Architecture specifies the **ISimpleFrameSite** interface to enable container controls. Containers may also support container controls without supporting **ISimpleFrameSite**, although the behavior cannot be guaranteed. For this reason, a component category exists for SimpleFrameSite controls where the full functionality of this interface is required.

In order to support container controls without implementing **ISimpleFrameSite**, an ActiveX Control Container must:

- Activate all controls at all times.
- Reparent the contained controls to the hWnd of the containing control.
- Remain the parent of the container control.

WS_GROUP and WS_TABSTOP Flags in Controls

A control should not use the WS_GROUP and WS_TABSTOP flags internally; some containers rely on these flags to manage keyboard handling.

Multiple Controls in One DLL

A single .OCX DLL can container any number of ActiveX Controls, thus simplifying the distribution and use of a set of related controls.

If you ship multiple controls in a single DLL, be sure to include the vendor name in each control name in the package. Including the vendors' names in each control name will enable users to easily identify controls within a package. For example, if you ship a DLL that implements three controls, Con1, Con2 and Con3, then the control names should be:

<Your company name> Con1 Control

<Your company name> Con2 Control

<Your company name> Con3 Control

The IOleContainer::EnumObjects Method

This method is used to enumerate over all the OLE objects contained in a document or form, returning an interface pointer for each OLE object. The container must return pointers to each OLE object that shares the same container. Nested forms or nested controls must also be enumerated.

Some containers implement extender controls, which wrap non-ActiveX Controls, and then return pointers to these extender controls as a form is enumerated. This behavior enables ActiveX Controls and ActiveX Control containers to integrate well with non-ActiveX Controls, and is thus recommended, but not required.

Enhanced Metafiles

Not surprisingly, enhanced metafiles provide more functionality than standard metafiles; using enhanced metafiles generally simplifies rendering code. An enhanced metafile DC is used in exactly the same way as a standard metafile DC. Enhanced metafiles are not available in 16-bit OLE. OLE supports enhanced metafiles, and includes backwards compatibility with standard metafiles and 16-bit applications.

32-bit ActiveX Control containers should use enhanced metafiles instead of standard metafiles.

Licensing

In order to embed licensed controls successfully, ActiveX Control containers must use **IClassFactory2** instead of **IClassFactory**. Several OLE creation and loading helper functions (i.e., **OleLoad** and **CoCreateInstance**) explicitly call **IClassFactory** and not **IClassFactory2**, and therefore cannot be used to create or load licensed ActiveX Controls. ActiveX Control Containers should explicitly create and load ActiveX Controls using **IClassFactory2**. In the future, Microsoft will update these standard APIs to use both **IClassFactory** and **IClassFactory2**, as appropriate.

Dual Interfaces

OLE Automation enables an object to expose a set of methods in two ways: via the **IDispatch** interface, and through direct OLE VTable binding. **IDispatch** is used by most tools available today, and offers support for late binding to properties and methods. VTable binding offers much higher performance because this method is called directly instead of through **IDispatch::Invoke**. **IDispatch** offers late bound support, where direct VTable binding offers a significant performance gain; both techniques are valuable and important in different scenarios. By labeling an interface as **dual** in the type library, an OLE Automation interface can be used either via **IDispatch**, or it can be bound to directly. Containers can thus choose the most appropriate technique. Support for dual interfaces is strongly recommended for both controls and containers.

IPropertyBag and IPersistPropertyBag

IPropertyBag and **IPersistPropertyBag** optimize save as text mechanisms, and therefore are recommended for ActiveX Control containers that implement a save as text mechanism. **IPropertyBag** is implemented by a container, and is roughly analogous to **IStream**. **IPersistPropertyBag** is implemented by controls, and is roughly analogous to **IPersistStream**.

Event Coordinate Translation

The 96 specification for controls requires that coordinates passed for events fired by the control change from being HIMETRIC to being Points based. This change brings the event passing of coordinates in line with properties and methods and thus coordinate translation is no longer the responsibility of the container. This raises certain compatibility issues where a control fires events using a coordinate base that it is not expecting, this should only be an issue where a 96 control container is hosting an older pre-96 control as follows:

- When an older pre-96 container hosts a 96 control the control will present the event coordinates as points, this should not cause the container any problems as the container should recognize the parameter type.
- When a 96 container hosts a pre-96 control the control will present the container with coordinates and expect the container to any translation necessary. However the 96 container will be expecting a control to conform to the 96 controls specification and present its coordinates as points. The control uses the **TranslateCoordinates** method on the **IOleControlSite** interface provided by the container in the same way as it does for properties and methods to achieve this.

As a result the user of a 96 container hosting pre-96 controls will need to be aware that further translation of coordinates may be necessary when events are fired.

Standard DISPIDS

A number of standard dispids have been defined for the 96 controls specification.

DISPID_MOUSEPOINTER

#define DISPID_MOUSEPOINTER -521

Property of type integer.

The Mousepointer property identifies standard mouse icons:

Value	Description
0	(Default) Shape determined by the object.
1	Arrow
2	Cross (cross-hair pointer)
3	I Beam
4	Icon (small square within a square)
5	Size (four-pointed arrow pointing north, south, east, and west)
6	Size NE SW (double arrow pointing northeast and southwest)
7	Size N S (double arrow pointing north and south)
8	Size NW, SE
9	Size E W (double arrow pointing east and west)
10	Up Arrow
11	Hourglass (wait)
12	No Drop
13	Arrow and hourglass
14	Arrow and question mark
15	Size all
99	Custom icon specified by the MouseIcon property

DISPID_MOUSEICON

#define DISPID_MOUSEICON -522

Property of type Picture.

DISPID_PICTURE

#define DISPID_PICTURE -523

Property of type picture.

DISPID_VALID

#define DISPID_VALID -524 // Is data in control valid?

Property of type BOOL.

Used to determine if the control has valid data or not.

DISPID_ AMBIENT_PALETTE

```
#define DISPID_AMBIENT_PALETTE          -726    // Container's HPAL
```

Used to allow the control to get the container's HPAL. If the container supplies an ambient palette then that is the only palette that may be realized into the foreground. Controls that wish to realize their own palettes must do so in the background. If their is no ambient palette provided by the container then the active control may realize its palette in the foreground. Palette handling is further discussed in Palette Behaviour for OLE Controls which is in the ActiveX SDK.

Databinding

A new databinding attribute has been added to allow properties distinguish between communicating changes only when focus leaves the control or during all property change notifications.

The new attribute known as **ImmediateBind** is to allow controls to differentiate two different types of bindable properties. One type of bindable property needs to notify every change to the database, for example with a checkbox control where every change needs to be sent through to the underlying database even though the control has not lost the focus. However controls such as a listbox only wish to have the change of a property notified to the database when the control loses focus, as the user may have changed the highlighted selection with the arrow keys before finding the desired setting, to have the change notification sent to the database every time that the user hit the arrow key would be give unacceptable performance. The new immediate bind property allows individual bindable properties on a form to have this behavior specified, when this bit is set all changes will be notified.

The new **ImmediateBind** bit maps through to the new VARFLAG_FIMMEDIATEBIND (0x80) and the FUNCFLAG_FIMMEDIATEBIND (0x80) bits in the VARFLAGS and FUNCFLAGS enumerations for the **ITypeInfo** interface allowing for the properties attributes to be inspected.

IAdviseSink Quick Info

The **IAdviseSink** interface enables containers and other objects to receive notifications of data changes, view changes, and compound-document changes occurring in objects of interest. Container applications, for example, require such notifications to keep cached presentations of their linked and embedded objects up-to-date. Calls to **IAdviseSink** methods are asynchronous, so the call is sent and then the next instruction is executed without waiting for the call's return.

For an advisory connection to exist, the object that is to receive notifications must implement **IAdviseSink**, and the objects in which it is interested must implement **IOleObject::Advise** and **IDataObject::DAdvise**. In-process objects and handlers may also implement **IViewObject::SetAdvise**. Objects implementing **IOleObject** must support all reasonable advisory methods. To simplify advisory notifications, OLE supplies implementations of the **IDataAdviseHolder** and **IOleAdviseHolder**, which keep track of advisory connections and send notifications to the proper sinks through pointers to their IAdviseSink interfaces. **IViewObject** (and its advisory methods) is implemented in the default handler.

As shown in the following table, an object that has implemented an advise sink registers its interest in receiving certain types of notifications by calling the appropriate method:

Call This Method	To Register for These Notifications
IOleObject::Advise	When a document is saved, closed, or renamed.
IDataObject::DAdvise	When a document's data changes.
IViewObject::SetAdvise	When a document's presentation changes.

When an event occurs that applies to a registered notification type, the object application calls the appropriate **IAdviseSink** method. For example, when an embedded object closes, it calls the **IAdviseSink::OnClose** method to notify its container. These notifications are asynchronous, occurring after the events that trigger them.

When to Implement

Objects, such as container applications and compound documents, implement **IAdviseSink** to receive notification of changes in data, presentation, name, or state of their linked and embedded objects. Implementers register for one or more types of notification, depending on their needs.

Notifications of changes to an embedded object originate in the server and flow to the container by way of the object handler. If the object is a linked object, the OLE link object intercepts the notifications from the object handler and notifies the container directly. All containers, the object handler, and the OLE link object register for OLE notifications. The typical container also registers for view notifications. Data notifications are usually sent to the OLE link object and object handler.

When to Use

Servers call the methods of **IAdviseSink** to notify objects with which they have an advisory connection of changes in an object's data, view, name, or state.

Note OLE does not permit synchronous calls in the implementation of asynchronous methods, so you cannot make synchronous calls within any of the the **IAdviseSink** interface's methods. For example, an implementation of **IAdviseSink::OnDataChange** cannot contain a call to **IDataObject::GetData**.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IAdviseSink Methods	Description
OnDataChange	Advises that data has changed.
OnViewChange	Advises that view of object has changed.
OnRename	Advises that name of object has changed.
OnSave	Advises that object has been saved to disk.
OnClose	Advises that object has been closed.

See Also

IAdviseSink2, IDataAdviseHolder, IOleAdviseHolder, IOleObject::Advise, IOleObject::Unadvise, IOleObject::EnumAdvise

IAdviseSink::OnClose Quick Info

Called by the server to notify all registered advisory sinks that the object has changed from the running to the loaded state.

Void OnClose();

Remarks

OnClose notification indicates that an object is making the transition from the running to the loaded state, so its container can take appropriate measures to ensure orderly shutdown. For example, an object handler must release its pointer to the object.

If the object that is closing is the last open object supported by its OLE server application, the application can also shut down.

In the case of a link object, the notification that the object is closing should always be interpreted to mean that the connection to the link source has broken.

IAdviseSink::OnDataChange Quick Info

Called by the server to notify a data object's currently registered advise sinks that data in the object has changed.

```
void OnDataChange(
    FORMATETC * pFormatetc,     //Pointer to format information
    STGMEDIUM * pStgmed         //Pointer to storage medium
);
```

Parameters

pFormatetc

[in] Pointer to the **FORMATETC** structure, which describes the format, target device, rendering, and storage information of the calling data object.

pStgmed

[in] Pointer to the **STGMEDIUM** structure, which defines the storage medium (global memory, disk file, storage object, stream object, GDI object, or undefined) and ownership of that medium for the calling data object.

Remarks

Object handlers and containers of link objects implement **IAdviseSink::OnDataChange** to take appropriate steps when notified that data in the object has changed. They also must call **IDataObject::DAdvise** to set up advisory connections with the objects in whose data they are interested. (See **IDataObject::DAdvise** for more information on how to specify an advisory connection for data objects.)

Containers that take advantage of OLE's caching support do not need to register for data-change notifications, because the information necessary to update the container's presentation of the object, including any changes in its data, are maintained in the object's cache.

Notes to Implementers

If you implement **IAdviseSink::OnDataChange** for a container, remember that this method is asynchronous and that making synchronous calls within asynchronous methods is not valid. Therefore, you cannot call **IDataObject::GetData** to obtain the data you need to update your object. Instead, you either post an internal message, or invalidate the rectangle for the changed data by calling **InvalidateRect** and waiting for a WM_PAINT message, at which point you are free to get the data and update the object.

The data itself, which is valid only for the duration of the call, is passed using the storage medium pointed to by *pmedium.* Since the caller owns the medium, the advise sink should not free it. Also, if *pmedium* points to an **IStorage** or **IStream** interface, the sink must not increment the reference count.

See Also

IDataObject::DAdvise

IAdviseSink::OnRename Quick Info

Called by the server to notify all registered advisory sinks that the object has been renamed.

```
Void OnRename(
  IMoniker * pmk      //Pointer to the new moniker of the object
);
```

Parameter

pmk

[in] Pointer to the **IMoniker** interface on the new full moniker of the object.

Remarks

OLE link objects normally implement **IAdviseSink::OnRename** to receive notification of a change in the name of a link source or its container. The object serving as the link source calls **OnRename** and passes its new full moniker to the object handler, which forwards the notification to the link object. In response, the link object must update its moniker. The link object, in turn, forwards the notification to its own container.

IAdviseSink::OnSave Quick Info

Called by the server to notify all registered advisory sinks that the object has been saved.

Void OnSave();

Remarks

Object handlers and link objects normally implement **IAdviseSink::OnSave** to receive notifications of when an object is saved to disk, either to its original storage (through a Save operation) or to new storage (through a Save As operation). Object Handlers and link objects register to be notified when an object is saved for the purpose of updating their caches, but then only if the advise flag passed during registration specifies ADVFCACHE_ONSAVE. Object handlers and link objects forward these notifications to their containers.

IAdviseSink::OnViewChange Quick Info

Notifies an object's registered advise sinks that its view has changed.

```
Void OnViewChange(
    DWORD dwAspect,     //Value specifying aspect of object
    LONG lindex         //Currently must be -1
);
```

Parameters

dwAspect

[in] The aspect, or view, of the object. Contains a value taken from the enumeration, **DVASPECT**.

lindex

[in] The portion of the view that has changed. Currently only -1 is valid.

Remarks

Containers register to be notified when an object's view changes by calling **IViewObject::SetAdvise**. Once registered, the object will call the sink's **IAdviseSink::OnViewChange** method when appropriate. **OnViewChange** can be called when the object is in either the loaded or running state.

Even though **DVASPECT** values are individual flag bits, *dwAspect* may represent only one value. That is, *dwAspect* cannot contain the result of an OR operation combining two or more **DVASPECT** values.

The *lindex* member represents the part of the aspect that is of interest. The value of *lindex* depends on the value of *dwAspect*. If *dwAspect* is either DVASPECT_THUMBNAIL or DVASPECT_ICON, *lindex* is ignored. If *dwAspect* is DVASPECT_CONTENT, *lindex* must be -1, which indicates that the entire view is of interest and is the only value that is currently valid.

See Also

IViewObject::SetAdvise

IAdviseSink2 Quick Info

The **IAdviseSink2** interface is an extension of **IAdviseSink**, adding the method **OnLinkSrcChange** to the contract to handle a change in the moniker of a linked object. This avoids overloading the implementation **IAdviseSink::OnRename** to handle the renaming of both embedded objects and linked objects. In applications where different blocks of code might execute depending on which of these two similar events has occurred, using the same method for both events complicates testing and debugging.

When to Implement

If your application supports links, you should definitely implement **IAdviseSink2**. Even if your application does not support links, it may do so in future releases. In general, your code would implement just **IAdviseSink2**, which, because of contract inheritance, must include implementations of all of the **IAdviseSink** methods, along with the single additional method of **IAdviseSink2**.

When to Use

When a link source is renamed, the link object should notify its container by calling **IAdviseSink::OnLinkSrcChange**. If you are using the default handler, containing the OLE link object, you can still have both embedded and linked objects call **IAdviseSink::OnRename**, but for linked objects, the OLE link object maps the notification to **IAdviseSink2::OnLinkSrcChange**.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IAdviseSink Methods	Description
OnDataChange	Advises that data has changed.
OnViewChange	Advises that view of object has changed.
OnRename	Advises that name of object has changed.
OnSave	Advises that object has been saved to disk.
OnClose	Advises that object has been closed.

IAdviseSink2 Method	Description
OnLinkSrcChange	Advises that link source has changed.

IAdviseSink2::OnLinkSrcChange Quick Info

Notifies the container that registered the advise sink that a link source has changed (either name or location), enabling the container to update the link's moniker.

```
void OnLinkSrcChange(

  IMoniker *pmk      //Pointer to the moniker of the new link source
);
```

Parameter

pmk

[in] Pointer to the **IMoniker** interface identifying the source of a linked object.

Remarks

A container of linked objects implements this method to receive notification in the event of a change in the moniker of its link source.

IAdviseSink2::OnLinkSrcChange is called by the OLE link object when it receives the **OnRename** notification from the link-source (object) application. The link object updates its moniker and sends the **OnLinkSrcChange** notification to containers that have implemented **IAdviseSink2**.

Notes to Implementers

Nothing prevents a link object from notifying its container of the moniker change by calling **IAdviseSink::OnRename** instead of **OnLinkSrcChange**. In practice, however, overloading **OnRename** to mean either that a link object's moniker has changed or that an embedded object's server name has changed makes it difficult for applications to determine which of these events has occurred. If the two events trigger different processing, as will often be the case, calling a different method for each makes the job of determining which event occurred much easier.

See Also

IAdviseSink::OnRename

IAdviseSinkEx Quick Info

The **IAdviseSinkEx** interface is derived from **IAdviseSink** to provide extensions for notifying the sink of changes in an object's view status.

When to Implement

Container applications and compound documents implement a site object with the **IAdviseSinkEx** interface to receive notification of changes in the view status of a contained object.

When to Use

A contained object, such as a control, calls the method of **IAdviseSinkEx** to notify its container of changes in its view status.

To determine which interface the sink supports, an object must call **QueryInterface** using the pointer that was passed to **IViewObject::SetAdvise**.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IAdviseSink Methods	Description
OnDataChange	Advises that data has changed.
OnViewChange	Advises that view of object has changed.
OnRename	Advises that name of object has changed.
OnSave	Advises that object has been saved to disk.
OnClose	Advises that object has been closed.

IAdviseSinkEx Methods	Description
OnViewStatusChange	Notifies the sink that a view status of an object has changed.

See Also

IAdviseSink, IViewObject::SetAdvise

IAdviseSinkEx::OnViewStatusChange Quick Info

Notifies the sink that a view status of an object has changed.

```
HRESULT OnViewStatusChange(
  DWORD dwViewStatus        //New view status
);
```

Parameters

dwViewStatus

[in] New view status specified in **VIEWSTATUS** enumeration values.

Return Values

S_OK

The sink was successfully notified of the new view status

Remarks

It is important that objects call the **IAdviseSink:OnViewChange** method whenever the object's view changes even when the object is in place active. Containers rely on this notification to keep an object's view up-to-date.

See Also

IAdviseSink:OnViewChange, **VIEWSTATUS**

IBindCtx Quick Info

The **IBindCtx** interface provides access to a bind context, which is an object that stores information about a particular moniker binding operation. You pass a bind context as a parameter when calling many methods of **IMoniker** and in certain functions related to monikers.

A bind context includes the following information:

- A **BIND_OPTS** structure containing a set of parameters that do not change during the binding operation. When a composite moniker is bound, each component uses the same bind context, so it acts as a mechanism for passing the same parameters to each component of a composite moniker.
- A set of pointers to objects that the binding operation has activated. The bind context holds pointers to these bound objects, keeping them loaded and thus eliminating redundant activations if the objects are needed again during subsequent binding operations.
- A pointer to the Running Object Table on the machine of the process that started the bind operation. Moniker implementations that need to access the Running Object Table should use the **IBindCtx::GetRunningObjectTable** method rather than using the **GetRunningObjectTable** function. This allows future enhancements to the system's **IBindCtx** implementation to modify binding behavior.
- A table of interface pointers, each associated with a string key. This capability enables moniker implementations to store interface pointers under a well-known string so that they can later be retrieved from the bind context. For example, OLE defines several string keys (e.g., "ExceededDeadline", "ConnectManually") that can be used to store a pointer to the object that caused an error during a binding operation.

When to Implement

You do not need to implement this interface. The system provides an **IBindCtx** implementation, accessible though a call to the **CreateBindCtx** function, that is suitable for all situations.

When to Use

Anyone writing a new moniker class by implementing the **IMoniker** interface must call **IBindCtx** methods in the implementation of several **IMoniker** methods. Moniker providers (servers that hand out monikers to identify their objects) may also need to call **IBindCtx** methods from their implementations of the **IOleItemContainer** or **IParseDisplayName** interfaces.

Moniker clients (objects that use monikers to acquire interface pointers to other objects) typically don't call many **IBindCtx** methods. Instead, they simply pass a bind context as a parameter in a call to an **IMoniker** method. To acquire an interface pointer and activate the indicated object (called binding to an object), moniker clients typically do the following:

1. Call the **CreateBindCtx** function to create a bind context and get a pointer to the **IBindCtx** interface on the bind context object..
2. If desired (although this is rarely necessary), the moniker client can call **IBindCtx::SetBindOptions** to specify the bind options.
3. Pass the bind context as a parameter to the desired **IMoniker** method (usually **IMoniker::BindToObject**).
4. Call **IUnknown::Release** on the bind context to release it.

Although applications that act as link containers (container applications that allow their documents to contain linked objects) are moniker clients, they rarely call **IMoniker** methods directly. Generally, they manipulate linked objects through the system implementation (in the default handler) of the **IOleLink** interface. This implementation calls the appropriate **IMoniker** methods as needed, and, in doing so,

passes pointers to **IBindCtx** interfaces on the proper bind context objects.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IBindCtx Methods	Description
RegisterObjectBound	Registers an object with the bind context.
RevokeObjectBound	Revokes an object's registration.
ReleaseBoundObjects	Releases all registered objects.
SetBindOptions	Sets the binding options.
GetBindOptions	Retrieves the binding options.
GetRunningObjectTable	Retrieves a pointer to the Running Object Table.
RegisterObjectParam	Associates an object with a string key.
GetObjectParam	Returns the object associated with a given string key.
EnumObjectParam	Enumerates all the string keys in the table.
RevokeObjectParam	Revokes association between an object and a string key.

See Also

CreateBindCtx, IMoniker, IOleItemContainer, IParseDisplayName

IBindCtx::EnumObjectParam Quick Info

Supplies a pointer to an **IEnumString** interface on an enumerator that can return the keys of the bind context's string-keyed table of pointers.

```
HRESULT EnumObjectParam(

  IEnumString **ppenum      //Indirect pointer to the enumerator object
);
```

Parameter

ppenum

[out] Indirect pointer to the **IEnumString** interface on the enumerator. If an error occurs, **ppenum* is set to NULL. If **ppenum* is non-NULL, the implementation calls **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

An enumerator was successfully created and the pointer supplied.

Remarks

This method provides an **IEnumString** pointer to an enumerator that can return the keys of the bind context's string-keyed table of pointers. The keys returned are the ones previously specified in calls to **IBindCtx::RegisterObjectParam**.

Notes to Callers

A bind context maintains a table of interface pointers, each associated with a string key. This enables communication between a moniker implementation and the caller that initiated the binding operation. One party can store an interface pointer under a string known to both parties so that the other party can later retrieve it from the bind context.

See Also

IBindCtx::RegisterObjectParam, **IEnumString**

IBindCtx::GetBindOptions Quick Info

Returns the binding options stored in this bind context.

```
HRESULT GetBindOptions(

  BIND_OPTS *pbindopts      //Pointer to a structure
);
```

Parameter

pbindopts

[in, out] Pointer to an initialized **BIND_OPTS** structure on entry that receives the current binding parameters on return.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The stored binding options were successfully returned.

Remarks

A bind context contains a block of parameters, stored in a **BIND_OPTS** structure, that are common to most **IMoniker** operations and that do not change as the operation moves from piece to piece of a composite moniker.

Notes to Callers

You typically call this method if you are writing your own moniker class (this requires that you implement the **IMoniker** interface). You call this method to retrieve the parameters specified by the moniker client.

You must initialize the **BIND_OPTS** structure that is filled in by this method. Before calling this method, you must initialize the *cbStruct* field of the structure to the size of the **BIND_OPTS** structure.

See Also

IBindCtx::SetBindOptions

IBindCtx::GetObjectParam

Quick Info

Retrieves the pointer associated with the specified key in the bind context's string-keyed table of pointers.

```
HRESULT GetObjectParam(
  LPOLESTR pszKey,     //Pointer to the key to be used
  IUnknown **ppunk     //Indirect pointer to the object associated with the key
);
```

Parameters

pszKey

[in] Pointer to a zero-terminated wide character string (two bytes per character) containing the key to search for. Key string comparison is case-sensitive.

ppunk

[out] When successful, indirect pointer to the **IUnknown** interface on the object associated with *pszKey*. In this case, the implementation calls **IUnknown::AddRef** on the parameter. It is the caller's responsibility to call **IUnknown::Release**. If an error occurs, *ppunk* is set to NULL.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The pointer associated with the specified key was successfully returned.

Remarks

A bind context maintains a table of interface pointers, each associated with a string key. This enables communication between a moniker implementation and the caller that initiated the binding operation. One party can store an interface pointer under a string known to both parties so that the other party can later retrieve it from the bind context.

The pointer this method retrieves must have previously been inserted into the table using the **IBindCtx::RegisterObjectParam** method.

Notes to Callers

Those writing a new moniker class (through an implementation of **IMoniker**) and some moniker clients (objects using a moniker to bind to an object) can call **IBindCtx::GetObjectParam**.

Objects using monikers to locate other objects can call this method when a binding operation fails to get specific information about the error that occurred. Depending on the error, it may be possible to correct the situation and retry the binding operation. See **IBindCtx::RegisterObjectParam** for more information.

Moniker implementations can call this method to deal with situations where a caller initates a binding operation and requests specific information. By convention, the implementer should use key names that begin with the string form of the CLSID of a moniker class (see the **StringFromCLSID** function).

See Also

IBindCtx::RegisterObjectParam, **IBindCtx::EnumObjectParam**

IBindCtx::GetRunningObjectTable Quick Info

Provides an interface pointer to the Running Object Table (ROT) for the machine on which this bind context is running.

```
HRESULT GetRunningObjectTable(
  IRunningObjectTable **pprot      //Indirect pointer to the Running Object Table
);
```

Parameter

pprot

[out] When successful, indirect pointer to the **IRunningObjectTable** interface on the Running Object Table. If an error occurs, **pprot* is set to NULL. If **pprot* is non-NULL, the implementation calls **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

A pointer to the ROT was returned successfully.

Remarks

The Running Object Table is a globally accessible table on each machine. It keeps track of all the objects that are currently running on the machine.

Notes to Callers

Typically, those implementing a new moniker class (through an implementation of **IMoniker** interface) call **IBindCtx::GetRunningObjectTable**. It is useful to call this method in an implementation of **IMoniker::BindToObject** or **IMoniker::IsRunning** to check whether a given object is currently running. You can also call this method in the implementation of **IMoniker::GetTimeOfLastChange** to learn when a running object was last modified.

Moniker implementations should call this method instead of using the **GetRunningObjectTable** function. This makes it possible for future implementations of **IBindCtx** to modify binding behavior.

See Also

IMoniker, **IRunningObjectTable**

IBindCtx::RegisterObjectBound Quick Info

Calls **IUnknown::AddRef** on the specified object to ensure that the object remains active until the bind context is released. The method stores a pointer to the object in the bind context's internal list of pointers.

```
HRESULT RegisterObjectBound(
  IUnknown *punk      //Pointer to the object being registered
);
```

Parameter

punk

[in] Pointer to the **IUnknown** interface on the object that is being registered as bound.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The object was successfully registered.

Remarks

Notes to Callers

Those writing a new moniker class (through an implementation of the **IMoniker** interface), should call this method whenever the implementation activates an object. This happens most often in the course of binding a moniker, but it can also happen while retrieving a moniker's display name, parsing a display name into a moniker, or retrieving the time that an object was last modified.

IBindCtx::RegisterObjectBound calls **IUnknown::AddRef** to create an additional reference to the object. You must, however, still release your own copy of the pointer. Note that calling this method twice for the same object creates two references to that object. You can release a reference obtained through a call to this method by calling **IBindCtx::RevokeObjectBound**. All references held by the bind context are released when the bind context itself is released.

Calling **IBindCtx::RegisterObjectBound** to register an object with a bind context keeps the object active until the bind context is released. Reusing a bind context in a subsequent binding operation (either for another piece of the same composite moniker, or for a different moniker) can make the subsequent binding operation more efficient because it doesn't have to reload that object. This, however, improves performance only if the subsequent binding operation requires some of the same objects as the original one, so you need to balance the possible performance improvement of reusing a bind context against the costs of keeping objects activated unnecessarily.

IBindCtx does not provide a method to retrieve a pointer to an object registered using **IBindCtx::RegisterObjectBound**. Assuming the object has registered itself with the Running Object Table, moniker implementations can call **IRunningObjectTable::GetObject** to retrieve a pointer to the object.

See Also

IBindCtx::ReleaseBoundObjects, **IBindCtx::RevokeObjectBound**, **IRunningObjectTable::GetObject**

IBindCtx::RegisterObjectParam Quick Info

Stores an **IUnknown** pointer on the specified object under the specified key in the bind context's string-keyed table of pointers. The method must call **IUnknown::AddRef** on the stored pointer.

```
HRESULT RegisterObjectParam(
  LPOLESTR pszKey,     //Pointer to the key to be used
  IUnknown *punk       //Pointer to the object to be associated with the key
);
```

Parameters

pszKey

[in] Pointer to a zero-terminated wide character string (two bytes per character) containing the key under which the object is being registered. Key string comparison is case-sensitive.

punk

[in] Pointer to the **IUnknown** interface on the object that is to be registered.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The pointer was successfully registered under the specified string.

Remarks

A bind context maintains a table of interface pointers, each associated with a string key. This enables communication between a moniker implementation and the caller that initiated the binding operation. One party can store an interface pointer under a string known to both parties so that the other party can later retrieve it from the bind context.

Binding operations subsequent to the use of this method can use **IBindCtx::GetObjectParam** to retrieve the stored pointer.

Notes to Callers

IBindCtx::RegisterObjectParam is useful to those implementing a new moniker class (through an implementation of **IMoniker**) and to moniker clients (those who use monikers to bind to objects).

In implementing a new moniker class, you call this method when an error occurs during moniker binding to inform the caller of the cause of the error. The key that you would obtain with a call to this method would depend on the error condition. The following lists common moniker binding errors, describing for each the keys that would be appropriate:

MK_E_EXCEEDEDDEADLINE

If a binding operation exceeds its deadline because a given object is not running, you should register the object's moniker using the first unused key from the list: "ExceededDeadline", "ExceededDeadline1", "ExceededDeadline2", etc. If the caller later finds the moniker in the Running Object Table, the caller can retry the binding operation.

MK_E_CONNECTMANUALLY

The "ConnectManually" key indicates a moniker whose binding requires assistance from the end user. The caller can retry the binding operation after showing the moniker's display name to request that the end user manually connect to the object. Common reasons for this error are that a password is needed or that a floppy needs to be mounted.

E_CLASSNOTFOUND

The "ClassNotFound" key indicates a moniker whose class could not be found (the server for the object identified by this moniker could not be located). If this key is used for an OLE compound-document object, the caller can use **IMoniker::BindToStorage** to bind to the object, and then try to carry out a Treat As... or Convert To... operation to associate the object with a different server. If this is successful, the caller can retry the binding operation.

If you're a moniker client with detailed knowledge of the implementation of the moniker you're using, you can also call this method to pass private information to that implementation.

You can define new strings as keys for storing pointers. By convention, you should use key names that begin with the string form of the CLSID of the moniker class (see the **StringFromCLSID** function).

If the *pszKey* parameter matches the name of an existing key in the bind context's table, the new object replaces the existing object in the table.

When you register an object using this method, the object is not released until one of the following occurs:

- It is replaced in the table by another object with the same key.
- It is removed from the table by a call to **IBindCtx::RevokeObjectParam**.
- The bind context is released. All registered objects are released when the bind context is released.

See Also

IBindCtx::GetObjectParam, **IBindCtx::RevokeObjectParam**, **IBindCtx::EnumObjectParam**

IBindCtx::ReleaseBoundObjects Quick Info

Releases all pointers to all objects that were previously registered by calls to **IBindCtx::RegisterObjectBound**.

HRESULT ReleaseBoundObjects(*void***);**

Return Value

S_OK

The objects were released successfully.

Remarks

You rarely call this method directly. The system's **IBindCtx** implementation calls this method when the pointer to the **IBindCtx** interface on the bind context is released (the bind context is released). If a bind context is not released, all of the registered objects remain active.

If the same object has been registered more than once, this method calls the **IUnknown::Release** method on the object the number of times it was registered.

See Also

IBindCtx::RegisterObjectBound

IBindCtx::RevokeObjectBound Quick Info

Releases the **IUnknown** pointer to the specified object and removes that pointer from the bind context's internal list of pointers. This undoes a previous call to **IBindCtx::RegisterObjectBound** for the same object.

```
HRESULT RevokeObjectBound(
  IUnknown *punk      //Pointer to the object whose registration is being revoked
);
```

Parameter

punk

[in] Pointer to the **IUnknown** interface on the object to be released.

Return Values

S_OK

The object was released successfully.

MK_E_NOTBOUND

Indicates that *punk* was not previously registered with a call to **IBindCtx::RegisterObjectBound**.

Remarks

You rarely call this method. This method is included for completeness.

See Also

IBindCtx::RegisterObjectBound

IBindCtx::RevokeObjectParam Quick Info

Removes the specified key and its associated pointer from the bind context's string-keyed table of objects. The key must have previously been inserted into the table with a call to **IBindCtx::RegisterObjectParam**.

```
HRESULT RevokeObjectParam(
  LPOLESTR pszKey      //Pointer to the key to be revoked
);
```

Parameter

pszKey

[in] Pointer to a zero-terminated wide character string (two bytes per character) containing the key to remove. Key string comparison is case-sensitive.

Return Values

S_OK

The specified key was successfully removed from the table.

S_FALSE

No object has been registered with the specified key.

Remarks

A bind context maintains a table of interface pointers, each associated with a string key. This enables communication between a moniker implementation and the caller that initiated the binding operation. One party can store an interface pointer under a string known to both parties so that the other party can later retrieve it from the bind context.

This method is used to remove an entry from the table. If the specified key is found, the bind context also releases its reference to the object.

See Also

IBindCtx::RegisterObjectParam

IBindCtx::SetBindOptions Quick Info

Specifies new values for the binding parameters stored in the bind context. Subsequent binding operations can call **IBindCtx::GetBindOptions** to retrieve the parameters.

```
HRESULT SetBindOptions(
  BIND_OPTS *pbindopts      //Pointer to a structure
);
```

Parameter

pbindopts

[in] Pointer to a **BIND_OPTS2** or a **BIND_OPTS** structure containing the binding parameters.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The parameters were stored successfully.

Remarks

A bind context contains a block of parameters, stored in a **BIND_OPTS2** or a **BIND_OPTS** structure, that are common to most **IMoniker** operations. These parameters do not change as the operation moves from piece to piece of a composite moniker.

Notes to Callers

This method can be called by moniker clients (those who use monikers to acquire interface pointers to objects).

When you first create a bind context using the **CreateBindCtx** function, the fields of the **BIND_OPTS** structure are initialized to the following values:

```
cbStruct = sizeof(BINDOPTS);
grfFlags = 0;
grfMode = STGM_READWRITE;
dwTickCountDeadline = 0;
```

You can use the **IBindCtx::SetBindOptions** method to modify these values before using the bind context, if you want values other than the defaults. See **BIND_OPTS** for more information.

> SetBindOptions only copies the struct members of BIND_OPTS2, but not the COSERVERINFO structure and the pointers it contains. Callers may not free any of these pointers until the bind context is released.

See Also

Bind_OPTS2, **IBindCtx::GetBindOptions**

IClassActivator

Specifies a method that retrieves a class object.

When to Implement

No implementation of a moniker or an object supporting **IClassActivator** currently exists within the system, however future versions of the operating system may contain such implementations. Implement the **IClassActivator** interface if you are writing a custom moniker type which you want to be able to compose to the left of a class moniker or any other moniker that supports binding to **IClassActivator**.

When to Use

Use **IClassActivator** if you write a custom moniker class that should behave similarly to class monikers when composed to the right of other monikers. File monikers also use this interface.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IClassActivator Method	Description
GetClassObject	Retrieves a class object.

IClassActivator::GetClassObject

Retrieves a class object. Similar to **CoGetClassObject**.

```
HRESULT GetClassObject(
  REFCLSID *pClassID      //CLSID of class object desired
  DWORD dwClsContext      //Values from CLSCTX
  LCID locale             //LCID constant
  REFIID riid             //IID of requested interface
  void ** ppv             //Indirect pointer to requested interface
);
```

Parameter

pClassID

[in] Points to the CLSID that Identifies the class whose class object is to be retrieved.

dwClsContext

[in] The context in which the class is expected to run; values are taken from the **CLSCTX** enumeration.

locale

[in] Any LCID constant as defined in WINNLS.H.

riid

[in] IID of the interface on the object to which a pointer is desired.

ppv

[out] On successful return, an indirect pointer to the requested interface.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The CLSID was successfully returned.

Remarks

This method returns the class identifier (CLSID) for an object, used in later operations to load object-specific code into the caller's context.

See Also

CoGetClassObject

IClassFactory Quick Info

The **IClassFactory** interface contains two methods intended to deal with an entire class of objects, and so is implemented on the class object for a specific class of objects (identified by a CLSID). The first method, **CreateInstance**, creates an uninitialized object of a specified CLSID, and the second, **LockServer**, locks the object's server in memory, allowing new objects to be created more quickly.

When to Implement

You must implement this interface for every class that you register in the system registry and to which you assign a CLSID, so objects of that class can be created.

When to Use

After calling the **CoGetClassObject** function to get an **IClassFactory** interface pointer to the class object, call the **CreateInstance** method of this interface to create a new uninitialized object.

It is not, however, always necessary to go through this process to create an object. To create a single uninitialized object, you can, instead, just call **CoCreateInstance**. OLE also provides numerous helper functions (with names of the form **OleCreate***Xxx*) to create compound document objects.

Call the **LockServer** method to keep the object server in memory and enhance performance only if you intend to create more than one object of the specified class.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IClassFactory Methods	**Description**
CreateInstance	Creates an uninitialized object.
LockServer	Locks object application open in memory.

See Also

CoGetClassObject, **CoCreateInstance**, **OleCreate**

IClassFactory::CreateInstance Quick Info

Creates an uninitialized object.

```
HRESULT CreateInstance(
  IUnknown * pUnkOuter,     //Pointer to whether object is or isn't part of an aggregate
  REFIID riid,              //Reference to the identifier of the interface
  void ** ppvObject         //Indirect pointer to the interface
);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, pointer to the controlling **IUnknown** interface of the aggregate. Otherwise, *pUnkOuter* must be NULL.

riid

[in] Reference to the identifier of the interface to be used to communicate with the newly created object. If *pUnkOuter* is NULL, this parameter is frequently the IID of the initializing interface; if *pUnkOuter* is non-NULL, *riid* must be **IID_IUnknown** (defined in the header as the IID for **IUnknown**).

ppvObject

[out] Indirect pointer to the requested interface. If the object does not support the interface specified in *riid*, *ppvObject* must be set to NULL.

Return Values

This method supports the standard return values E_UNEXPECTED, E_OUTOFMEMORY, and E_INVALIDARG, as well as the following:

S_OK

The specified object was created.

CLASS_E_NOAGGREGATION

The *pUnkOuter* parameter was non-NULL and the object does not support aggregation.

E_NOINTERFACE

The object that *ppvObject* points to does not support the interface identified by *riid*.

Remarks

The **IClassFactory** interface is always on a class object. The **CreateInstance** method creates an uninitialized object of the class identified with the specified CLSID. When an object is created in this way, the CLSID must be registered in the system registry with **CoRegisterClassObject**.

The *pUnkOuter* parameter indicates whether the object is being created as part of an aggregate. Object definitions are not required to support aggregation – they must be specifically designed and implemented to support it.

The *riid* parameter specifies the IID (interface identifier) of the interface through which you will communicate with the new object. If *pUnkOuter* is non-NULL (indicating aggregation), the value of the *riid*

parameter must be **IID_IUnknown**. If the object is not part of an aggregate, *riid* often specifies the interface though which the object will be initialized.

For OLE embeddings, the initialization interface is **IPersistStorage**, but in other situations, other interfaces are used. To initialize the object, there must be a subsequent call to an appropriate method in the initializing interface. Common initialization functions include **IPersistStorage::InitNew** (for new, blank embeddable components), **IPersistStorage::Load** (for reloaded embeddable components), **IPersistStream::Load**, (for objects stored in a stream object) or **IPersistFile::Load** (for objects stored in a file).

In general, if an application supports only one class of objects, and the class object is registered for single use, only one object can be created. The application must not create other objects, and a request to do so should return an error from **IClassFactory::CreateInstance**. The same is true for applications that support multiple classes, each with a class object registered for single use; a **CreateInstance** for one class followed by a **CreateInstance** for any of the classes should return an error.

To avoid returning an error, applications that support multiple classes with single-use class objects can revoke the registered class object of the first class by calling **CoRevokeClassObject** when a request for instantiating a second is received. For example, suppose there are two classes, A and B. When **IClassFactory::CreateInstance** is called for class A, revoke the class object for B. When B is created, revoke the class object for A. This solution complicates shutdown because one of the class objects might have already been revoked (and cannot be revoked twice).

See Also

CoRegisterClassObject, **CoRevokeClassObject**, **CoCreateInstance**, **CoGetClassObject**

IClassFactory::LockServer Quick Info

Called by the client of a class object to keep a server open in memory, allowing instances to be created more quickly.

```
HRESULT LockServer(
  BOOL fLock      //Increments or decrements the lock count
);
```

Parameter

fLock

[in] If **TRUE**, increments the lock count; if **FALSE**, decrements the lock count.

Return Values

This method supports the standard return values E_FAIL, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The specified object was either locked (*fLock* = TRUE) or unlocked from memory (*fLock* = FALSE).

Remarks

IClassFactory::LockServer controls whether an object's server is kept in memory. Keeping the application alive in memory allows instances to be created more quickly.

Notes to Callers

Most clients do not need to call this function. It is provided only for those clients that require special performance in creating multiple instances of their objects.

Notes to Implementers

If the lock count is zero, there are no more objects in use, and the application is not under user control, the server can be closed. One way to implement **IClassFactory::LockServer** is to call **CoLockObjectExternal**.

The process that locks the object application is responsible for unlocking it. Once the class object is released, there is no mechanism that guarantees the caller connection to the same class later (as in the case where a class object is registered as single-use). It is important to count all calls, not just the last one, to **IClassFactory::LockServer**, because calls must be balanced before attempting to release the pointer to the **IClassFactory** interface on the class object or an error results. For every call to **LockServer** with *fLock* set to TRUE, there must be a call to **LockServer** with *fLock* set to FALSE. When the lock count and the class object reference count are both zero, the class object can be freed.

See Also

CoLockObjectExternal

IClassFactory2 Quick Info

The **IClassFactory2** interface enables a class factory object, in any sort of object server, to control object creation through licensing. This interface is an extension to **IClassFactory**. This extension enables a class factory executing on a licensed machine to provide a license key that can be used later to create an object instance on an unlicensed machine. Such considerations are important for objects like controls that are used to build applications on a licensed machine. Subsequently, the application built must be able to run on an unlicensed machine. The license key gives only that one client application the right to instantiate objects through **IClassFactory2** when a full machine license does not exist.

When to Implement

Implement this interface on a class factory object if you need to control object creation through a license. A class that supports licensing should be marked in an object's type information with the **[licensed]** attribute on the object's **coclass** entry.

The **CreateInstance** method inherited from **IClassFactory** is allowed to return CLASS_E_NOTLICENSED to indicate that object creation is controlled through licensing. The caller can create an instance of this object only through **IClassFactory2::CreateInstanceLic** if the caller has a license key obtained from **IClassFactory2::RequestLicKey**. Otherwise, no object creation is allowed.

When to Use

Use this interface to create licensed objects or to obtain a license key that can be used in later creations.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IClassFactory Methods	Description
CreateInstance	Creates an uninitialized object.
LockServer	Locks object application open in memory.

IClassFactory2 Methods	Description
GetLicInfo	Fills a **LICINFO** structure with information on the licensing capabilities of this class factory.
RequestLicKey	Creates and returns a license key that the caller can save and use later in calls to **IClassFactory2::CreateInstanceLic**.
CreateInstanceLic	Creates an instance of the licensed object given a license key from **IClassFactory2::RequestLicKey**.

See Also

IClassFactory

IClassFactory2::CreateInstanceLic Quick Info

Creates an instance of the object class supported by this class factory, given a license key previously obtained from **IClassFactory2::RequestLicKey**. This method is the only possible means to create an object on an otherwise unlicensed machine.

```
HRESULT CreateInstanceLic(
  IUnknown* pUnkOuter ,          //Pointer to controlling unknown of aggregated object
  IUnknown* pUnkReserved ,       //Unused. Must be NULL.
  REFIID riid ,                  //Reference to the identifier of the interface
  BSTR bstrKey ,                 //License key provided by IClassFactory2::RequestLicKey
  void** ppvObject               //Indirect pointer to the interface of the type specified in riid
);
```

Parameters

pUnkOuter

[in] Pointer to the controlling **IUnknown** interface on the outer unknown if this object is being created as part of an aggregate. If the object is not part of an aggregate, this parameter must be NULL.

pUnkReserved

[in] Unused. Must be NULL.

riid

[in] Reference to the identifier of the interface to be used to communicate with the newly created object.

bstrKey

[in] Run-time license key previously obtained from **IClassFactory2::RequestLicKey** that is required to create an object.

ppvObject

[out] Indirect pointer to the interface of the type specified in *riid*. This parameter is set to NULL on failure.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The license was successfully created.

E_NOTIMPL

This method is not implemented because objects can only be created on fully licensed machines through **IClassFactory::CreateInstance**.

E_POINTER

The pointers passed in *bstrKey* or *ppvObject* are not valid. For example, it may be NULL.

E_NOINTERFACE

The object can be created (and the license key is valid) except the object does not support the interface specified by *riid*.

CLASS_E_NOAGGREGATION

The *pUnkOuter* parameter is non-NULL, but this object class does not support aggregation.

CLASS_E_NOTLICENSED

The key provided in *bstrKey* is not a valid license key.

Remarks

Notes to Implementers

If the class factory does not provide a license key (that is, **IClassFactory2::RequestLicKey** returns E_NOTIMPL and the *fRuntimeKeyAvail* field in **LICINFO** is set to FALSE in **IClassFactory2::GetLicInfo**), then this method can also return E_NOTIMPL. In such cases, the class factory is implementing **IClassFactory2** simply to specify whether or not the machine is licensed at all through the *fLicVerified* field of **LICINFO**.

See Also

IClassFactory2::GetLicInfo, **IClassFactory2::RequestLicKey**, **LICINFO**

IClassFactory2::GetLicInfo Quick Info

Fills a caller-allocated **LICINFO** structure with information describing the licensing capabilities of this class factory.

```
HRESULT GetLicInfo(

  LICINFO* pLicInfo      //Pointer to the structure
);
```

Parameters

pLicInfo

[out] Pointer to the caller-allocated **LICINFO** structure to be filled on output.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The **LICINFO** structure was successfully filled in.

E_POINTER

The address in *pLicInfo* is not valid. For example, it may be NULL.

Remarks

Notes to Implementers

E_NOTIMPL is not allowed as a return value since this method provides critical information for the client of a licensed class factory.

See Also

IClassFactory2::CreateInstanceLic, **IClassFactory2::RequestLicKey**, **LICINFO**

IClassFactory2::RequestLicKey Quick Info

If the *fRuntimeKeyAvail* field in **LICINFO** has been returned as TRUE from **IClassFactory2::GetLicInfo**, then this method creates and returns a license key. The caller can save the license key persistently and use it later in calls to **IClassFactory2::RequestLicKey**.

```
HRESULT RequestLicKey(
    DWORD dwReserved ,      //Unused. Must be zero.
    BSTR* pbstrKey          //Pointer to the license key
);
```

Parameters

dwReserved

[in] Unused. Must be zero.

pbstrKey

[out] Pointer to the caller-allocated **BSTR** variable that receives the callee-allocated license key on successful return from this method. This parameter is set to NULL on any failure.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The license key was successfully created.

E_NOTIMPL

This class factory does not support run-time license keys.

E_POINTER

The address in *pbstrKey* is not valid. For example, it may be NULL.

CLASS_E_NOTLICENSED

This class factory supports run-time licensing, but the current machine itself is not licensed. Thus, a run-time key is not available on this machine.

Remarks

The caller can save the license key for subsequent calls to **IClassFactory2::CreateInstanceLic** to create objects on an otherwise unlicensed machine.

Notes to Callers

The caller must free the **BSTR** with **SysFreeString** when the key is no longer needed. The value of *fRuntimeKeyAvail* is returned through a previous call to **IClassFactory2::GetLicInfo**.

Notes to Implementers

This method allocates the **BSTR** key with `SysAllocString` or `SysAllocString[Len]`, and the caller becomes responsible for this **BSTR** once this method returns successfully.

This method need not be implemented when a class factory does not support run-time license keys.

See Also

IClassFactory2::CreateInstanceLic, **IClassFactory2::GetLicInfo**, **LICINFO**

IClientSecurity

Quick Info

Gives the client control over the call-security of individual interfaces on a remote object.

All proxies generated by the COM MIDL compiler support the **IClientSecurity** interface automatically. If a call to **QueryInterface** for **IClientSecurity** fails, either the object is implemented in-process or it is remoted by a custom marshaler which does not support security (a custom marshaler may support security by offering the **IClientSecurity** interface to the client). The proxies passed as parameters to an **IClientSecurity** method must be from the same object as the **IClientSecurity** interface. That is, each object has a distinct **IClientSecurity** interface: calling **IClientSecurity** on one object and passing a proxy to another object will not work.

When to Implement

The system proxy manager provides an implementation to objects, so you would typically not implement this interface.

If, however, you are defining objects that support custom marshaling, you may choose to implement **IClientSecurity** on the objects' custom proxy to maintain a consistent programming model for the objects' client applications. You may also choose to support this interface on in-process objects.

When to Use

Call the methods of this interface to examine or modify the security settings of a particular connection to an out-of-process object. For example, you might temporarily establish a higher security level – one with complex encryption – only for the period when sensitive information or data is being sent to the object. Alternately, you might establish different proxies to the same object with different security levels and use them to support different clients that are calling your object, or to support different operations within your application.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IClientSecurity Methods	Description
IClientSecurity::QueryBlanket	Retrieves authentication information.
IClientSecurity::SetBlanket	Sets the authentication information that will be used to make calls on the specified proxy.
IClientSecurity::CopyProxy	Makes a copy of the specified proxy.

See Also

Security in COM

IClientSecurity::CopyProxy Quick Info

Makes a private copy of the specified proxy.

```
HRESULT CopyProxy(
  IUnknown *   punkProxy ,     //IUnknown pointer to the proxy to copy
  IUnknown **  ppunkCopy       //Indirect IUnknown pointer to the copy
);
```

Parameter

punkProxy

[in] Points to the **IUnknown** interface on the proxy to be copied. May not be NULL.

ppunkCopy

[out] On successful return, points to the location of the **IUnknown** pointer to the copy of the proxy. It may not be NULL.

Return Values

S_OK

Success.

E_INVALIDARG

One or more arguments are invalid.

Remarks

IClientSecurity::CopyProxy makes a private copy of the specified proxy for the calling client. Its authentication information may be changed through a call to **IClientSecurity::SetBlanket** without affecting any other clients of the original proxy. The copy has the default values for the authentication information. The copy has one reference and must be released.

The helper function **CoCopyProxy** encapsulates a **QueryInterface** call on the proxy for a pointer to **IClientSecurity**, and with that pointer calls **IClientSecurity::CopyProxy**, and then releases the **IClientSecurity** pointer.

Local interfaces may not be copied. **IUnknown** and **IClientSecurity** are examples of existing local interfaces.

Copies of the same proxy have a special relationship with respect to **QueryInterface**. Given a proxy, *a*, of the *IA* interface of a remote object, suppose a copy of *a* is created, called *b*. In this case, calling QueryInterface from the *b* proxy for IID_IA will not retrieve the IA interface on *b*, but the one on *a*, the original proxy with the "default" security settings for the *IA* interface.

See Also

CoCopyProxy

IClientSecurity::QueryBlanket Quick Info

Retrieves authentication information.

```
HRESULT QueryBlanket(
  void*   pProxy ,                          //Location for the proxy to query
  DWORD*   pAuthnSvc ,                      //Location for the current authentication service
  DWORD*   pAuthzSvc ,                      //Location for the current authorization service
  OLECHAR **   pServerPrincName ,           //Location for the current principal name
  DWORD * pAuthnLevel ,                     //Location for the current authentication level
  DWORD * pImpLevel ,                       //Location for the current impersonation level
  RPC_AUTH_IDENTITY_HANDLE **   ppAuthInfo , //Location for the value passed to IClientSecurity::SetBlanket
  DWORD **   pCapabilities                  //Location for flags indicating further capabilities of the proxy
);
```

Parameter

pProxy

[in] Pointer to the proxy to query.*pAuthnSvc*

[out] Pointer to a DWORD value defining the current *authentication* service. This will be a single value taken from the list of **RPC_C_AUTHN_*xxx*** constants. May be NULL, in which case the current authentication service is not retrieved.

pAuthzSvc

[out] Pointer to a DWORD value defining the current *authorization* service. This will be a single value taken from the list of **RPC_C_AUTHZ_*xxx*** constants. May be NULL, in which case the current authorization service is not retrieved.

pServerPrincName

[out] Pointer to the current principal name. The string will be allocated by the one called using **CoTaskMemAlloc** and must be freed by the caller using **CoTaskMemFree** when they are done with it. May be NULL, in which case the principal name is not retrieved.

pAuthnLevel

[out] Pointer to a DWORD value defining the current authentication level. This will be a single value taken from the list of **RPC_C_AUTHN_LEVEL_*xxx*** constants. May be NULL, in which case the current authentication level is not retrieved.

pImpLevel

[out] Pointer to a DWORD value defining the current impersonation level. This will be a single value taken from the list of **RPC_C_IMP_LEVEL_*xxx*** constants. May be NULL, in which case the current authentication level is not retrieved.

ppAuthInfo

[out] Pointer to the pointer value passed to **IClientSecurity::SetBlanket** indicating the identity of the client. Because this points to the value itself and is not a copy, it should not be manipulated. May be NULL, in which case the information is not retrieved.

pCapabilities

[out] Pointer to a DWORD of flags indicating further capabilities of the proxy. Currently, no flags are defined for this parameter and it will only return zero. May be NULL, in which case the flags indicating

further capabilities are not retrieved.

Return Values

S_OK

Success.

E_INVALIDARG

One or more arguments are invalid.

E_OUTOFMEMORY

Insufficient memory to create the *pServerPrincName* out-parameter.

Remarks

IClientSecurity::QueryBlanket is called by the client to retrieve the authentication information COM will use on calls made from the specified proxy. With a pointer to an interface on the proxy, the client would first call **QueryInterface** for a pointer to **IClientSecurity**, then, with this pointer, would call **IClientSecurity::QueryBlanket**, followed by releasing the pointer. This sequence of calls is encapsulated in the helper function **CoQueryProxyBlanket**.

In *pProxy*, you can pass any proxy, such as a proxy you get through a call to **CoCreateInstance**, **CoUnmarshalInterface**, or just passing an interface pointer as a parameter. It can be any interface. You cannot pass a pointer to something that is not a proxy. Thus you can't pass a pointer to an interface that has the local keyword in its interface definition since no proxy is created for such an interface. **IUnknown** is the exception.

See Also

CoQueryProxyBlanket

IClientSecurity::SetBlanket Quick Info

Sets the authentication information that will be used to make calls on the specified proxy.

```
HRESULT SetBlanket(
  void *   pProxy ,                                  //Indicates the proxy to set
  DWORD    dwAuthnSvc ,                              //Authentication service to use
  DWORD    dwAuthzSvc ,                              //Authorization service to use
  WCHAR *  pServerPrincName ,                        //The server principal name to use with the authentication service
  DWORD    dwAuthnLevel ,                            //The authentication level to use
  DWORD    dwImpLevel ,                              //The impersonation level to use
  RPC_AUTH_IDENTITY_HANDLE *   pAuthInfo ,           //The identity of the client
  DWORD    dwCapabilities                            //Undefined – capability flags
);
```

Parameter

pProxy

[in] Indicates the proxy to set.

dwAuthnSvc

[in] A single DWORD value from the list of **RPC_C_AUTHN_*xxx*** constants indicating the *authentication* service to use. It may be RPC_C_AUTHN_NONE if no authentication is required. RPC_C_AUTHN_WINNT is the only value available on NT by default.

dwAuthzSvc

[in] A single DWORD value from the list of **RPC_C_AUTHZ_*xxx*** constants indicating the *authorization* service to use. If you are using the NT authentication service, use RPC_C_AUTHZ_NONE.

pServerPrincName

[in] Pointer to a WCHAR string that indicates the server principal name to use with the authentication service. If you are using RPC_C_AUTHN_WINNT, the principal name must be NULL.

dwAuthnLevel

[in] A single DWORD value from the list of **RPC_C_AUTHN_LEVEL_*xxx*** constants indicating the authentication level to use.

dwImpLevel

[in] A single DWORD value from the list of **RPC_C_IMP_LEVEL_*xxx*** constants indicating the impersonation level to use. Currently, only RPC_C_IMP_LEVEL_IMPERSONATE and RPC_C_IMP_LEVEL_IDENTIFY are supported by NTLMSSP.

pAuthInfo

[in] Pointer to an RPC_AUTH_IDENTITY_HANDLE value that establishes the identity of the client. It is authentication-service specific. Some authentication services allow the application to pass in a different user name and password. COM keeps a pointer to the memory passed in until COM is uninitialized or a new value is set. If NULL is specified COM uses the current identity (the process token). For NTLMSSP the structure is SEC_WINNT_AUTH_IDENTITY_W. The format of this structure depends on the provider of the authentication service.

dwCapabilities

[in] A DWORD defining flags to establish indicating the further capabilities of this proxy. Currently, no capability flags are defined.

The caller should specify EOAC_NONE. EOAC_MUTUAL_AUTH is defined and may be used by other security providers, but is not supported by NTLMSSP. Thus, NTLMSSP will accept this flag without generating an error but without providing mutual authentication.

Return Values

S_OK

Success, append the headers.

E_INVALIDARG

One or more arguments is invalid.

Remarks

IClientSecurity::SetBlanket sets the authentication information that will be used to make calls on the specified proxy. The values specified here override the values chosen by automatic security. Calling this method changes the security values for all other users of the specified proxy. Call **IClientSecurity::CopyProxy** to make a private copy of the proxy.

By default, COM will choose the first available authentication service and authorization service available on both the client and server machines and the principal name which the server registered for that authentication service. Currently, COM will not try another authentication service if the first fails.

If *pAuthInfo* is NULL, it defaults to the current process token. *dwAuthnLevel* and *dwImpLevel* default to the values specified to **CoInitializeSecurity**. If **CoInitializeSecurity** is not called, the defaults are taken from the registry. The initial value for dwAuthnLevel on a proxy will be the higher of the value set on the client's call to **CoInitializeSecurity** and the server's call to **CoInitializeSecurity** .

Security information cannot be set on local interfaces. For example, it is illegal to set security on the **IClientSecurity** interface. However, since that interface is supported locally, there is no need for security. **IUnknown** is a special case. There are several cases. First, **IUnknown** cannot be copied. Thus all users of an object get the same security. Second, **SetBlanket** can be used to set the security used for calls to **QueryInterface**. However, since **QueryInterface** is heavily cached, the server might not see the call. Third, **AddRef** and **Release** always use the security set with **CoInitializeSecurity**, never the values set with **SetBlanket**.

See Also

CoSetProxyBlanket, **CoQueryProxyBlanket**, **RPC_C_AUTHN_xxx**, **RPC_C_AUTHZ_*xxx***, **RPC_C_AUTHN_LEVEL_*xxx***, **RPC_C_IMP_LEVEL_*xxx***

IConnectionPoint Quick Info

The **IConnectionPoint** interface supports connection points for connectable objects.

Connectable objects support the following features:

- Outgoing interfaces, such as event sets
- The ability to enumerate the IIDs of the outgoing interfaces
- The ability to connect and disconnect sinks to the object for those outgoing IIDs
- The ability to enumerate the connections that exist to a particular outgoing interface

When to Implement

Implement this interface as part of support for connectable objects. To create a connectable object, you need to implement objects that provide four related interfaces:

- **IConnectionPointContainer**
- **IEnumConnectionPoints**
- **IConnectionPoint**
- **IEnumConnections**

The **IConnectionPointContainer** interface is implemented on the connectable object to indicate the existence of the outgoing interfaces. It provides access to an enumerator sub-object with the **IEnumConnectionPoints** interface. It also provides access to all the connection point sub-objects, each of which implements the **IConnectionPoint** interface. The **IConnectionPoint** interface provides access to an enumerator sub-object with the **IEnumConnections** interface.

Each connection point is a separate sub-object to avoid circular reference counting problems.

A connection point controls how many connections (one or more) it will allow in its implementation of **IConnectionPoint::Advise**. A connection point that allows only one interface can return E_NOTIMPL from the **IConnectionPoint::EnumConnections** method.

When to Use

A client can use the **IConnectionPointContainer** interface:

- To obtain access to an enumerator sub-object with the **IEnumConnectionPoints** interface. The **IEnumConnectionPoints** interface can then be used to enumerate connection points for each outgoing IID.
- To obtain access to connection point sub-objects with the **IConnectionPoint** interface for each outgoing IID. Through the **IConnectionPoint** interface, a client starts or terminates an advisory loop with the connectable object and the client's own sink. The client can also use the **IConnectionPoint** interface to obtain an enumerator object with the **IEnumConnections** interface to enumerate the connections that it knows about.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IConnectionPoint Methods	Description
GetConnectionInterface	Returns the IID of the outgoing interface managed by this connection point.
GetConnectionPointContainer	Returns the parent (connectable) object's **IConnectionPointContainer** interface pointer.
Advise	Creates a connection between a connection point and a client's sink, where the sink implements the outgoing interface supported by this connection point.
Unadvise	Terminates a notification previously set up with **Advise**.
EnumConnections	Returns an object to enumerate the current advisory connections for this connection point.

See Also

IConnectionPoint, IConnectionPointContainer, IEnumConnectionPoints, IEnumConnections

IConnectionPoint::Advise Quick Info

Establishes a connection between the connection point object and the client's sink.

```
HRESULT Advise(
  IUnknown *pUnk ,        //Pointer to the client's advise sink
  DWORD *pdwCookie        //Pointer to the connection point identifier used by Unadvise
);
```

Parameter

pUnk

[in] Pointer to the **IUnknown** interface on the client's advise sink. The client's sink receives outgoing calls from the connection point.

pdwCookie

[out] Pointer to a returned token that uniquely identifies this connection. The caller uses this token later to delete the connection by passing it to the **IConnectionPoint::Unadvise** method. If the connection was not successfully established, this value is zero.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The connection has been established and **pdwCookie* has the connection token.

E_POINTER

The value in *pUnk* or *pdwCookie* is not valid. For example, either pointer may be NULL.

CONNECT_E_ADVISELIMIT

The connection point has already reached its limit of connections and cannot accept any more.

CONNECT_E_CANNOTCONNECT

The sink does not support the interface required by this connection point.

Remarks

Advise establishes a connection between the connection point and the caller's sink identified with *pUnk*.

The connection point must call `pUnk->QueryInterface(iid, ...)` to obtain the correct outgoing interface pointer to call when events occur, where *iid* is the IID for the outgoing interface managed by the connection point. When *iid* is passed to the **IConnectionPointContainer::FindConnectionPoint** method, an interface pointer to this same connection point is returned.

Notes to Implementers

The connection point must query the *pUnk* pointer for the correct outgoing interface. If this query fails, this method must return CONNECT_E_CANNOTCONNECT.

The *pdwCookie* value must be unique for each connection to any given instance of a connection point.

See Also

IConnectionPoint::Unadvise

IConnectionPoint::EnumConnections Quick Info

Creates an enumerator object to iterate through the current connections for this connection point.

```
HRESULT EnumConnections(
  IEnumConnections **ppEnum      //Indirect pointer to the newly created enumerator
);
```

Parameters

ppEnum

[out] Indirect pointer to the **IEnumConnections** interface on the newly created enumerator.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The enumerator object was successfully created.

E_POINTER

The address in *ppEnum* is not valid. For example, it may be NULL.

E_NOTIMPL

The connection point does not support enumeration.

Remarks

Notes to Callers

The caller is responsible for calling `(*ppEnum)->Release` when the enumerator is no longer needed.

See Also

IEnumConnections

IConnectionPoint::GetConnectionInterface Quick Info

Returns the IID of the outgoing interface managed by this connection point.

```
HRESULT GetConnectionInterface(
  IID *pIID      //Pointer to an IID variable
);
```

Parameters

pIID

[out] Pointer to the identifier of the outgoing interface managed by this connection point.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The caller's variable *pIID* contains the identifier of the outgoing interface managed by this connection point.

E_POINTER

The address in *pIID* is not valid. For example, it may be NULL.

Remarks

Using the **IEnumConnectionPoints** interface, a client can obtain a pointer to the **IConnectionPoint** interface. Using that pointer and the **GetConnectionInterface** method, the client can determine the IID of each connection point enumerated. The IID returned from this method must enable the caller to access this same connection point through **IConnectionPointContainer::FindConnectionPoint**.

Notes to Implementers

This method must be implemented in any connection point; E_NOTIMPL is not an acceptable return value.

See Also

IConnectionPoint, **IConnectionPointContainer::FindConnectionPoint**, **IEnumConnectionPoints**

IConnectionPoint::GetConnectionPointContainer Quick Info

Retrieves the **IConnectionPointContainer** interface pointer for the parent connectable object.

```
HRESULT GetConnectionPointContainer(
  IConnectionPointContainer **ppCPC      //Indirect pointer
);
```

Parameters

ppCPC

[out] Indirect pointer to the **IConnectionPointContainer** interface on the parent connectable object.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The **IConnectionPointContainer** pointer was successfully returned.

E_POINTER

The value in *ppCPC* is not a valid interface pointer. For example, it may be NULL.

Remarks

Notes to Callers

This method calls **IConnectionPointContainer::AddRef**. The caller is responsible for calling **IConnectionPointContainer::Release** to release this pointer when done.

Notes to Implementers

This method must call **IConnectionPointContainer::AddRef** before returning.

This method must be implemented in any connection point; E_NOTIMPL is not an acceptable return value.

See Also

IConnectionPoint, **IConnectionPointContainer**

IConnectionPoint::Unadvise Quick Info

Terminates an advisory connection previously established through **IConnectionPoint::Advise**. The *dwCookie* parameter identifies the connection to terminate.

```
HRESULT Unadvise(
  DWORD dwCookie      //Connection token
);
```

Parameters

dwCookie

[in] Connection token previously returned from **IConnectionPoint::Advise**.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The connection was successfully terminated.

CONNECT_E_NOCONNECTION

The value in *dwCookie* does not represent a valid connection.

Remarks

When an advisory connection is terminated, the connection point calls the **Release** method on the pointer that was saved for the connection during the **IConnectionPoint::Advise** method. This **Release** reverses the **AddRef** that was performed during the **IConnectionPoint::Advise** when the connection point calls the advisory sink's **QueryInterface**.

See Also

IConnectionPoint::Advise

IConnectionPointContainer Quick Info

The **IConnectionPointContainer** interface supports connection points for connectable objects.

Connectable objects support the following features:

- Outgoing interfaces, such as event sets
- The ability to enumerate the IIDs of the outgoing interfaces
- The ability to connect and disconnect sinks to the object for those outgoing IIDs
- The ability to enumerate the connections that exist to a particular outgoing interface

When to Implement

Implement this interface as part of support for connectable objects. To support connectable objects, you need to provide four related interfaces:

- **IConnectionPointContainer**
- **IEnumConnectionPoints**
- **IConnectionPoint**
- **IEnumConnections**

The **IConnectionPointContainer** interface indicates the existence of the outgoing interfaces. It provides access to an enumerator sub-object with the **IEnumConnectionPoints** interface. It also provides a connection point sub-object with the **IConnectionPoint** interface. The **IConnectionPoint** interface provides access to an enumerator sub-object with the **IEnumConnections** interface.

The connection point is a separate sub-object to avoid circular reference counting problems.

Implement the **IConnectionPointContainer** interface to make a connectable object, that is, an object with outgoing interfaces.

When to Use

Through **IConnectionPointContainer**, you can locate a specific connection point for one IID or obtain an enumerator to enumerate the connections points.

Use the **IConnectionPointContainer** interface to obtain access to:

- Enumerator sub-objects with the **IEnumConnectionPoints** interface. The **IEnumConnectionPoints** interface can then be used to enumerate connection points for each outgoing IID.
- Connection point sub-objects with the **IConnectionPoint** interface for each IID. Through the **IConnectionPoint** interface, a client starts or terminates an advisory loop with the connectable object and the client's own sink. The client can also use the **IConnectionPoint** interface to obtain an enumerator object with the **IEnumConnections** interface to enumerate the connections it knows about.

When to Use

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.

Release	Decrements reference count.

IConnectionPointContainer Methods	Description
EnumConnectionPoints	Returns an object to enumerate all the connection points supported in the connectable object.
FindConnectionPoint	Returns a pointer to the **IConnectionPoint** interface for a specified connection point.

See Also

IConnectionPoint, **IEnumConnectionPoints**, **IEnumConnections**

IConnectionPointContainer::EnumConnectionPoints Quick Info

Creates an enumerator object to iterate through all the connection points supported in the connectable object, one connection point per outgoing IID.

```
HRESULT EnumConnectionPoints(
  IEnumConnectionPoints **ppEnum      //Indirect pointer to the newly created enumerator
);
```

Parameters

ppEnum

[out] Indirect pointer to the **IEnumConnectionPoints** interface on the newly created enumerator.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The enumerator was successfully created.

E_POINTER

The value in *ppEnum* is not valid. For example, it may be NULL.

Remarks

Since **IEnumConnectionPoints** enumerates pointers to **IConnectionPoint**, the caller must use **IConnectionPoint::GetConnectionInterface** to determine the interface identifer of the outgoing interface that the connection point supports.

Notes to Callers

The caller is responsible for releasing the enumerator by calling `(*ppEnum)->Release` when it is no longer needed.

Notes to Implementers

Returning E_NOTIMPL is specifically disallowed because, with the exception of type information, there would be no other means through which a caller could find the IIDs of the outgoing interfaces. Since a connectable object typically has a fixed set of known outgoing interfaces, it is straightforward to implement the enumerator on top of a fixed length array of IIDs known at compile time.

See Also

IEnumConnectionPoints

IConnectionPointContainer::FindConnectionPoint Quick Info

Returns a pointer to the **IConnectionPoint** interface of a connection point for a specified IID, if that IID describes a supported outgoing interface.

```
HRESULT FindConnectionPoint(
  REFIID riid ,                  //Requested connection point's interface identifier
  IConnectionPoint **ppCP        //Indirect pointer to the variable of the requested IID
);
```

Parameters

riid

[in] Interface identifier of the outgoing interface whose connection point object is being requested.

ppCP

[out] Indirect pointer to the **IConnectionPoint** interface on the connection point that supports *riid*. This parameter is set to NULL on failure of the call.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The *ppCP* pointer has a valid interface pointer.

E_POINTER

The address in *ppCP* is not valid. For example, it may be NULL.

CONNECT_E_NOCONNECTION

This connectable object does not support the outgoing interface specified by *riid*.

Remarks

This method is the **QueryInterface** equivalent for an object's outgoing interfaces, where the outgoing interface is specified with *riid* and where the interface pointer returned is always that of a connection point.

Notes to Callers

If the call is successful, the caller is responsible for releasing the connection point by calling `(*ppCP)->Release` when the connection point is no longer needed.

Notes to Implementers

E_NOTIMPL is not allowed as a return value for this method. Any implementation of **IConnectionPointContainer** must implement this method for the connectable object's outgoing interfaces.

See Also

IConnectionPoint

IDataAdviseHolder Quick Info

The **IDataAdviseHolder** interface contains methods that create and manage advisory connections between a data object and one or more advise sinks. Its methods are intended to be used to implement the advisory methods of **IDataObject**. **IDataAdviseHolder** is implemented on an advise holder object. Its methods establish and delete data advisory connections and send notification of change in data from a data object to an object that requires this notification, such as an OLE container, which must contain an advise sink.

Advise sinks are objects that require notification of change in the data the object contains and implement the **IAdviseSink** interface. Advise sinks are also usually associated with OLE compound document containers.

When to implement

Typically, you use the OLE-provided implementation of **IDataAdviseHolder** to simplify your implementation of the **DAdvise**, **DUnadvise**, and **EnumDAdvise** methods in the **IDataObject** interface, and to send notification of data change as appropriate. It would be necessary to implement **IDataAdviseHolder** only in the case where there may be a need for a custom data advise holder object, whose methods are to be used to implement the **IDataObject** methods in a set of servers.

When to use

Your implementation of the advisory methods of **IDataObject** can call the methods in **IDataAdviseHolder**. The first time you receive a call to **IDataObject::DAdvise**, call the function **CreateDataAdviseHolder** to create an instance of the OLE-provided advise holder and get a pointer to its **IDataAdviseHolder** interface. Then, in implementing the **IDataObject** interface on the data object, you delegate implementations of the **DAdvise**, **DUnadvise**, and **EnumDAdvise** methods to the corresponding methods in **IDataAdviseHolder**.

When the data of interest to an advise sink actually changes, you call **IDataAdviseHolder::SendOnDataChange** from the data object to carry out the necessary notifications.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IDataAdviseHolder Methods	Description
Advise	Creates a connection between an advise sink and a data object so the advise sink can receive notification of change in the data object.
Unadvise	Destroys a notification connection previously set up with the **Advise** method.
EnumAdvise	Returns an object that can be used to enumerate the current advisory connections.
SendOnDataChange	Sends a change notification back

to each advise sink that is currently being managed.

See Also

IDataObject, **IAdviseSink**

IDataAdviseHolder::Advise Quick Info

Creates a connection between an advise sink and a data object for receiving notifications.

```
HRESULT Advise(
  IDataObject * pDataObject,      //Pointer to the data object for which notifications are requested
  FORMATETC * pFormatetc,         //Pointer to the description of data to the advise sink
  DWORD advf,                     //Flags that specify how the notification takes place
  IAdviseSink * pAdvSink,         //Pointer to the advise sink requesting notification
  DWORD * pdwConnection           //Pointer to the connection token
);
```

Parameters

pDataObject

[in] Pointer to the **IDataObject** interface on the data object for which notifications are requested. If data in this object changes, a notification is sent to the advise sinks that have requested notification.

pFormatetc

[in] Pointer to the specified format, medium, and target device that is of interest to the advise sink requesting notification. For example, one sink may want to know only when the bitmap representation of the data in the data object changes. Another sink may be interested in only the metafile format of the same object. Each advise sink is notified when the data of interest changes. This data is passed back to the advise sink when notification occurs.

advf

[in] Contains a group of flags for controlling the advisory connection. Valid values are from the enumeration **ADVF**. However, only some of the possible **ADVF** values are relevant for this method. The following table briefly describes the relevant values; a more detailed description can be found in the description of the **ADVF** enumeration.

ADVF Value	Description
ADVF_NODATA	Asks that no data be sent along with the notification.
ADVF_ONLYONCE	Causes the advisory connection to be destroyed after the first notification is sent. An implicit call to **IDataAdviseHolder::Unadvise** is made on behalf of the caller to remove the connection.
ADVF_PRIMEFIRST	Causes an initial notification to be sent regardless of whether or not data has changed from its current state.
ADVF_DATAONSTOP	When specified with ADVF_NODATA, this flag causes a last notification with the data included to be sent before the data object is destroyed. When ADVF_NODATA is not specified, this flag has no effect.

pAdvSink

[in] Pointer to the **IAdviseSink** interface on the advisory sink that receives the change notification.

pdwConnection

[out] Pointer to a DWORD token that identifies this connection. The calling object can later delete the advisory connection by passing this token to **IDataAdviseHolder::Unadvise**. If this value is zero, the connection was not established.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The advisory connection was created.

Remarks

Through the connection established through this method, the advisory sink can receive future notifications in a call to **IAdviseSink::OnDataChange**.

An object issues a call to **IDataObject::DAdvise** to request notification on changes to the format, medium, or target device of interest. This data is specified in the *pFormatetc* parameter. The **DAdvise** method is usually implemented to call **IDataAdviseHolder::Advise** to delegate the task of setting up and tracking a connection to the advise holder. When the format, medium, or target device in question changes, the data object calls **IDataAdviseHolder::SendOnDataChange** to send the necessary notifications.

The established connection can be deleted by passing the value in *pdwConnection* in a call to **IDataAdviseHolder::Unadvise**.

See also

ADVF, **CreateDataAdviseHolder**, **FORMATETC**, **IDataAdviseHolder::Unadvise**, **IDataObject::DAdvise**

IDataAdviseHolder::EnumAdvise Quick Info

Returns a pointer to an **IEnumStatdata** interface on an enumeration object that can be used to enumerate the current advisory connections.

```
HRESULT EnumAdvise(

  IEnumSTATDATA ** ppenumAdvise      //Indirect pointer on the new enumerator object
);
```

Parameter

ppenumAdvise

[out] Indirect pointer to the **IEnumStatdata** interface on the new enumerator object. If this value is NULL, there are no connections to advise sinks at this time.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The enumerator object is successfully instantiated or there are no connections.

Remarks

This method must supply a pointer to an implementation of the **IEnumSTATDATA** interface, one of the standard enumerator interfaces that contain the **Next**, **Reset**, **Clone**, and **Skip** methods, on an enumerator object. Its methods allow you to enumerate the data stored in an array of **STATDATA** structures. You get a pointer to the OLE implementation of **IDataAdviseHolder** through a call to **CreateDataAdviseHolder**, and then call **IDataAdviseHolder::EnumAdvise** to implement **IDataObject::EnumDAdvise**.

Adding more advisory connections while the enumerator object is active has an undefined effect on the enumeration that results from this method.

See Also

IEnum*XXXX*, **IEnumSTATDATA**, **IDataObject::EnumDAdvise**

IDataAdviseHolder::SendOnDataChange Quick Info

Sends notifications to each advise sink for which there is a connection established by calling the **IAdviseSink::OnDataChange** method for each advise sink currently being handled by this instance of the advise holder object.

```
HRESULT SendOnDataChange(
    IDataObject * pDataObject,      //Pointer to the data object that has changed
    DWORD dwReserved,               //Reserved
    DWORD advf                      //Advise flags
);
```

Parameters

pDataObject

[in] Pointer to the **IDataObject** interface on the data object in which the data has just changed. This pointer is used in subsequent calls to **IAdviseSink::OnDataChange**.

dwReserved

[in] Reserved for future use; must be zero.

advf

[in] Container for advise flags that specify how the call to **IAdviseSink::OnDataChange** is made. These flag values are from the enumeration **ADVF**. Typically, the value for *advf* is NULL. The only exception occurs when the data object is shutting down and must send a final notification that includes the actual data to sinks that have specified ADVF_DATAONSTOP and ADVF_NODATA in their call to **IDataObject::DAdvise**. In this case, *advf* contains ADVF_DATAONSTOP.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The call to **IAdviseSink::OnDataChange** was made.

Remarks

The data object must call this method when it detects a change that would be of interest to an advise sink that has previously requested notification.

Most notifications include the actual data with them. The only exception is if the ADVF_NODATA flag was previously specified when the connection was initially set up in the **IDataAdviseHolder::Advise** method.

Before calling the **IAdviseSink::OnDataChange** method for each advise sink, this method obtains the actual data by calling the **IDataObject::GetData** method through the pointer specified in the *pDataObject* parameter.

See Also

ADVF, **IAdviseSink::OnDataChange**

IDataAdviseHolder::Unadvise Quick Info

Removes a connection between a data object and an advisory sink that was set up through a previous call to **IDataAdviseHolder::Advise**. **IDataAdviseHolder::Unadvise** is typically called in the implementation of **IDataObject::DUnadvise**.

```
HRESULT Unadvise(
  DWORD dwConnection      //Connection to remove
);
```

Parameter

dwConnection

[in] DWORD token that specifies the connection to remove. This value was returned by **IDataAdviseHolder::Advise** when the connection was originally established.

Return Values

S_OK

The specified connection was successfully deleted.

OLE_E_NOCONNECTION

The specified *dwConnection* is not a valid connection.

See Also

IDataAdviseHolder::Advise, **IDataObject::DUnadvise**

IDataObject Quick Info

The **IDataObject** interface specifies methods that enable data transfer and notification of changes in data. Data transfer methods specify the format of the transferred data along with the medium through which the data is to be transferred. Optionally, the data can be rendered for a specific target device. In addition to methods for retrieving and storing data, the **IDataObject** interface specifies methods for enumerating available formats and managing connections to advisory sinks for handling change notifications.

The term "data object" is used to mean any object that supports an implementation of the **IDataObject** interface. Implementations vary, depending on what the data object is required to do; in some data objects, the implementation of certain methods not supported by the object could simply be the return of E_NOTIMPL. For example, some data objects do not allow callers to send them data. Other data objects do not support advisory connections and change notifications. However, for those data objects that do support change notifications, OLE provides an object called a data advise holder. An interface pointer to this holder is available through a call to the helper function **CreateDataAdviseHolder**. A data object can have multiple connections, each with its own set of attributes. The OLE data advise holder simplifies the task of managing these connections and sending the appropriate notifications.

When to Implement

Implement the **IDataObject** interface if you are developing a container or server application that is capable of transferring data. For example, if your application allows its data to be pasted or dropped into another application, you must implement the **IDataObject** interface. OLE compound document object servers that support objects that can be embedded or linked must implement **IDataObject**.

OLE provides implementations in its default object handler and its cache.

When to Use

Any object that can receive data calls the methods in the **IDataObject** interface.

When you call the data transfer methods in the **IDataObject** interface, you specify a format, a medium, and, optionally, a target device for which the data should be rendered. Objects, such as containers, that want to be notified through their advise sinks when the data in the data object changes call the **IDataObject** advisory methods to set up an advisory connection through which notifications can be sent.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IDataObject Methods	Description
GetData	Renders the data described in a FORMATETC structure and transfers it through the STGMEDIUM structure.
GetDataHere	Renders the data described in a **FORMATETC** structure and transfers it through the **STGMEDIUM** structure allocated by the caller.
QueryGetData	Determines whether the data

	object is capable of rendering the data described in the **FORMATETC** structure.
GetCanonicalFormatEtc	Provides a potentially different but logically equivalent **FORMATETC** structure.
SetData	Provides the source data object with data described by a **FORMATETC** structure and an **STGMEDIUM** structure.
EnumFormatEtc	Creates and returns a pointer to an object to enumerate the **FORMATETC** supported by the data object.
DAdvise	Creates a connection between a data object and an advise sink so the advise sink can receive notifications of changes in the data object.
DUnadvise	Destroys a notification previously set up with the **DAdvise** method.
EnumDAdvise	Creates and returns a pointer to an object to enumerate the current advisory connections.

IDataObject::DAdvise Quick Info

Called by an object supporting an advise sink to create a connection between a data object and the advise sink. This enables the advise sink to be notified of changes in the data of the object.

```
HRESULT DAdvise(
  FORMATETC * pFormatetc,      //Pointer to data of interest to the advise sink
  DWORD advf,                  //Flags that specify how the notification takes place
  IAdviseSink * pAdvSink,      //Pointer to the advise sink
  DWORD * pdwConnection        //Pointer to a token that identifies this connection
);
```

Parameters

pFormatetc

[in] Pointer to a **FORMATETC** structure that defines the format, target device, aspect, and medium that will be used for future notifications. For example, one sink may want to know only when the bitmap representation of the data in the the data object changes. Another sink may be interested in only the metafile format of the same object. Each advise sink is notified when the data of interest changes. This data is passed back to the advise sink when notification occurs.

advf

[in] DWORD that specifies a group of flags for controlling the advisory connection. Valid values are from the enumeration **ADVF**. However, only some of the possible **ADVF** values are relevant for this method. The following table briefly describes the relevant values. Refer to **ADVF** for a more detailed description.

ADVF Value	Description
ADVF_NODATA	Asks the data object to avoid sending data with the notifications. Typically data is sent. This flag is a way to override the default behavior. When ADVF_NODATA is used, the TYMED member of the **STGMEDIUM** structure that is passed to OnDataChange will usually contain TYMED_NULL. The caller can then retrieve the data with a subsequent **IDataObject::GetData** call.
ADVF_ONLYONCE	Causes the advisory connection to be destroyed after the first change notification is sent. An implicit call to **IDataObject::DUnadvise** is made on behalf of the caller to remove the connection.
ADVF_PRIMEFIRST	Asks for an additional initial notification. The combination of ADVF_ONLYONCE and ADVF_PRIMEFIRST provides, in effect, an asynchronous **IDataObject::GetData** call.
ADVF_DATAONSTOP	When specified with ADVF_NODATA, this flag causes a last notification with the data included to to be sent before the

data object is destroyed.

If used without ADVF_NODATA, **IDataObject::DAdvise** can be implemented in one of the following ways:

- the ADVF_DATAONSTOP can be ignored.
- the object can behave as if ADVF_NODATA was specified.
- a change notification is sent only in the shutdown case. Data changes prior to shutdown do not cause a notification to be sent.

pAdvSink

[in] Pointer to the **IAdviseSink** interface on the advisory sink that will receive the change notification.

pdwConnection

[out] Pointer to a DWORD token that identifies this connection. You can use this token later to delete the advisory connection (by passing it to **IDataObject::DUnadvise**). If this value is zero, the connection was not established.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK

The advisory connection was created.

E_NOTIMPL

This method is not implemented on the data object.

DV_E_LINDEX

Invalid value for *lindex*; currently, only -1 is supported.

DV_E_FORMATETC

Invalid value for *pFormatetc*.

OLE_E_ADVISENOTSUPPORTED

The data object does not support change notification.

Remarks

IDataObject::DAdvise creates a change notification connection between a data object and the caller. The caller provides an advisory sink to which the notifications can be sent when the object's data changes.

Objects used simply for data transfer typically do not support advisory notifications and return OLE_E_ADVISENOTSUPPORTED from **IDataObject::DAdvise**.

Notes to Callers

The object supporting the advise sink calls **IDataObject::DAdvise** to set up the connection, specifying the format, aspect, medium, and/or target device of interest in the **FORMATETC** structure passed in. If the data object does not support one or more of the requested attributes or the sending of notifications at all, it can refuse the connection by returning OLE_E_ADVISENOTSUPPORTED.

Containers of linked objects can set up advisory connections directly with the bound link source or indirectly through the standard OLE link object that manages the connection. Connections set up with the bound link source are not automatically deleted. The container must explicitly call **IDataObject::DUnAdvise** on the bound link source to delete an advisory connection. The OLE link object, manipulated through the **IOleLink** interface, is implemented in the default handler. Connections set up through the OLE link object are destroyed when the link object is deleted.

The OLE default link object creates a "wildcard advise" with the link source so OLE can maintain the time of last change. This advise is specifically used to note the time that anything changed. OLE ignores all data formats that may have changed, noting only the time of last change. To allow wildcard advises, set the **FORMATETC** members as follows before calling **IDataObject::DAdvise**:

```
cf == 0;
ptd == NULL;
dwAspect ==-1;
lindex == -1
tymed == -1;
```

The advise flags should also include ADVF_NODATA. Wildcard advises from OLE should always be accepted by applications.

Notes to Implementers

To simplify the implementation of **DAdvise** and the other notification methods in **IDataObject** (**DUnadvise** and **EnumAdvise**) that supports notification, OLE provides an advise holder object that manages the registration and sending of notifications. To get a pointer to this object, call the helper function **CreateDataAdviseHolder** on the first invocation of **DAdvise**. This supplies a pointer to the object's **IDataAdviseHolder** interface. Then, delegate the call to the **IDataAdviseHolder::Advise** method in the data advise holder, which creates, and subsequently manages, the requested connection.

See Also

ADVF, **FORMATETC**, **CreateDataAdviseHolder**, **IAdviseSink::OnDataChange**, **IDataObject::DUnAdvise**

IDataObject::DUnadvise Quick Info

Destroys a notification connection that had been previously set up.

```
HRESULT DUnadvise(
  DWORD dwConnection      //Connection to remove
);
```

Parameter

dwConnection

[in] DWORD token that specifies the connection to remove. Use the value returned by **IDataObject::DAdvise** when the connection was originally established.

Return Values

S_OK

The specified connection was successfully deleted.

OLE_E_NOCONNECTION

The specified *dwConnection* is not a valid connection.

OLE_E_ADVISENOTSUPPORTED

This **IDataObject** implementation does not support notification.

Remarks

This methods destroys a notification created with a call to the **IDataObject::DAdvise** method.

If the advisory connection being deleted was initially set up by delegating the **IDataObject::DAdvise** call to **IDataAdviseHolder::Advise**, you must delegate this call to **IDataAdviseHolder::Unadvise** to delete it.

See Also

IDataObject::DAdvise, **IDataAdviseHolder::Unadvise**

IDataObject::EnumDAdvise

Quick Info

Creates an object that can be used to enumerate the current advisory connections.

```
HRESULT EnumDAdvise(
  IEnumSTATDATA ** ppenumAdvise      //Indirect pointer
);
```

Parameter

ppenumAdvise

[out] Indirect pointer to the **IEnumSTATDATA** interface on the new enumerator object. If the supplied value is NULL, there are no connections to advise sinks at this time.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The enumerator object is successfully instantiated or there are no connections.

OLE_E_ADVISENOTSUPPORTED

Advisory notifications are not supported by this object.

Remarks

The enumerator object created by this method implements the **IEnumSTATDATA** interface, one of the standard enumerator interfaces that contain the **Next**, **Reset**, **Clone**, and **Skip** methods. **IEnumSTATDATA** permits the enumeration of the data stored in an array of **STATDATA** structures. Each of these structures provides information on a single advisory connection, and includes **FORMATETC** and **ADVF** information, as well as the pointer to the advise sink and the token representing the connection.

Notes to Callers

After getting a pointer through this method, the data object can call the appropriate enumeration methods. While the enumeration is in progress, the effect of adding more advisory connections on the subsequent enumeration is undefined.

Notes to Implementers

It is recommended that you use the OLE data advise holder object to handle advisory connections. With the pointer obtained through a call to **CreateDataAdviseHolder**, implementing **IDataObject::EnumDAdvise** becomes a simple matter of delegating the call to **IDataAdviseHolder::EnumAdvise**. This creates the enumerator and supplies the pointer to the OLE implementation of **IEnumSTATDATA**. At that point, you can call its methods to enumerate the current advisory connections.

See Also

IEnumSTATDATA, **IDataAdviseHolder::EnumAdvise**

IDataObject::EnumFormatEtc Quick Info

Creates an object for enumerating the **FORMATETC** structures for a data object. These structures are used in calls to **IDataObject::GetData** or **IDataObject::SetData**.

```
HRESULT EnumFormatEtc(
  DWORD dwDirection,                  //Specifies a value from the enumeration DATADIR
  IEnumFORMATETC ** ppenumFormatetc   //Indirect pointer to the new enumerator object
);
```

Parameters

dwDirection

[in] Direction of the data through a value from the enumeration **DATADIR**.

```
typedef enum tagDATADIR
{
    DATADIR_GET = 1,
    DATADIR_SET = 2,
} DATADIR;
```

The value DATADIR_GET enumerates the formats that can be passed in to a call to **IDataObject::GetData**. The value DATADIR_SET enumerates those formats that can be passed in to a call to **IDataObject::SetData**.

ppenumFormatetc

[out] Indirect pointer to the **IEnumFORMATETC** interface on the new enumerator object.

Return Values

This method supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

Enumerator object was successfully created.

E_NOTIMPL

The direction specified by *dwDirection* is not supported.

OLE_S_USEREG

Requests that OLE enumerate the formats from the registry.

Remarks

IDataObject::EnumFormatEtc creates an enumerator object that can be used to determine all of the ways the data object can describe data in a **FORMATETC** structure, and supplies a pointer to its **IEnumFORMATETC** interface. This is one of the standard enumerator interfaces.

Notes to Callers

Having obtained the pointer, the caller can enumerate the FORMATETC structures by calling the enumeration methods of **IEnumFORMATETC**. Because the formats can change over time, there is no

guarantee that an enumerated format is currently supported because the formats can change over time. Accordingly, applications should treat the enumeration as a hint of the format types that can be passed. The caller is responsible for calling **IEnumFormatEtc::Release** when it is finished with the enumeration.

IDataObject::EnumFormatEtc is called when one of the following actions occurs:

- An application calls **OleSetClipboard**. OLE must determine what data to place on the Clipboard and whether it is necessary to put OLE 1 compatibility formats on the Clipboard.
- Data is being pasted from the Clipboard or dropped. An application uses the first acceptable format.
- The Paste Special dialog box is displayed. The target application builds the list of formats from the **FORMATETC** entries.

Notes to Implementers

Formats can be registered statically in the registry or dynamically during object initialization. If an object has an unchanging list of formats and these formats are registered in the registry, OLE provides an implementation of a FORMATETC enumeration object that can enumerate formats registered under a specific CLSID in the registry. A pointer to its **IEnumFORMATETC** interface is available through a call to the helper function **OleRegEnumFormatEtc**. In this situation, therefore, you can implement the **EnumFormatEtc** method simply with a call to this function.

EXE applications can effectively do the same thing by implementing the method to return the value OLE_S_USEREG. This return value instructs the default object handler to call **OleRegEnumFormatEtc**. Object applications that are implemented as DLL object applications cannot return OLE_S_USEREG, so must call **OleRegEnumFormatEtc** directly.

Private formats can be enumerated for OLE 1 objects, if they are registered with the RequestDataFormats or SetDataFormats keys in the registry. Also, private formats can be enumerated for OLE objects (all versions after OLE 1), if they are registered with the GetDataFormats or SetDataFormats keys.

For OLE 1 objects whose servers do not have RequestDataFormats or SetDataFormats information registered in the registry, a call to **IDataObject::EnumFormatEtc** passing DATADIR_GET only enumerates the Native and Metafile formats, regardless of whether they support these formats or others. Calling **EnumFormatEtc** passing DATADIR_SET on such objects only enumerates Native, regardless of whether the object supports being set with other formats.

The **FORMATETC** structure returned by the enumeration usually indicates a NULL target device (*ptd*). This is appropriate because, unlike the other members of **FORMATETC**, the target device does not participate in the object's decision as to whether it can accept or provide the data in either a **SetData** or **GetData** call.

The TYMED member of **FORMATETC** often indicates that more than one kind of storage medium is acceptable. You should always mask and test for this by using a Boolean OR operator.

See Also

FORMATETC, **OleRegEnumFormatEtc**, **IEnumFormatEtc**, **IDataObject::SetData**, **IDataObject::GetData**

IDataObject::GetCanonicalFormatEtc Quick Info

Provides a standard **FORMATETC** structure that is logically equivalent to one that is more complex. You use this method to determine whether two different **FORMATETC** structures would return the same data, removing the need for duplicate rendering.

```
HRESULT GetCanonicalFormatEtc(
  FORMATETC * pFormatetcIn,      //Pointer to the FORMATETC structure
  FORMATETC * pFormatetcOut      //Pointer to the canonical equivalent FORMATETC structure
);
```

Parameters

pFormatetcIn

[in] Pointer to the FORMATETC structure that defines the format, medium, and target device that the caller would like to use to retrieve data in a subsequent call such as **IDataObject::GetData**. The TYMED member is not significant in this case and should be ignored.

pFormatetcOut

[out] Pointer to a **FORMATETC** structure that contains the most general information possible for a specific rendering, making it canonically equivalent to *pFormatetcIn*. The caller must allocate this structure and the **GetCanonicalFormatEtc** method must fill in the data. To retrieve data in a subsequent call like **IDataObject::GetData**, the caller uses the supplied value of *pFormatetcOut*, unless the value supplied is NULL. This value is NULL if the method returns DATA_S_SAMEFORMATETC. The TYMED member is not significant in this case and should be ignored.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK

The returned **FORMATETC** structure is different from the one that was passed.

DATA_S_SAMEFORMATETC

The **FORMATETC** structures are the same and NULL is returned in *pFormatetcOut*.

DV_E_LINDEX

Invalid value for *lindex*; currently, only -1 is supported.

DV_E_FORMATETC

Invalid value for *pFormatetc*.

OLE_E_NOTRUNNING

Object application is not running.

Remarks

If a data object can supply exactly the same data for more than one requested **FORMATETC** structure, **IDataObject::GetCanonicalFormatEtc** can supply a "canonical", or standard FORMATETC that gives

the same rendering as a set of more complicated FORMATETC structures. For example, it is common for the data returned to be insensitive to the target device specified in any one of a set of otherwise similar FORMATETC structures.

Notes to Callers

A call to this method can determine whether two calls to **IDataObject::GetData** on a data object, specifying two different FORMATETC structures, would actually produce the same renderings, thus eliminating the need for the second call and improving performance. If the call to **GetCanonicalFormatEtc** results in a canonical format being written to the *pFormatetcOut* parameter, the caller then uses that structure in a subsequent call to **IDataObject::GetData**.

Notes to Implementers

Conceptually, it is possible to think of FORMATETC structures in groups defined by a canonical FORMATETC that provides the same results as each of the group members. In constructing the canonical FORMATETC, you should make sure it contains the most general information possible that still produces a specific rendering.

For data objects that never provide device-specific renderings, the simplest implementation of this method is to copy the input **FORMATETC** to the output **FORMATETC**, store a NULL in the *ptd* field of the output **FORMATETC**, and return DATA_S_SAMEFORMATETC.

See Also

IDataObject::GetData, **FORMATETC**

IDataObject::GetData Quick Info

Called by a data consumer to obtain data from a source data object. The **GetData** method renders the data described in the specified **FORMATETC** structure and transfers it through the specified **STGMEDIUM** structure. The caller then assumes responsibility for releasing the **STGMEDIUM** structure.

```
HRESULT GetData(
  FORMATETC * pFormatetc,     //Pointer to the FORMATETC structure
  STGMEDIUM * pmedium         //Pointer to the STGMEDIUM structure
);
```

Parameters

pFormatetc

[in] Pointer to the FORMATETC structure that defines the format, medium, and target device to use when passing the data. It is possible to specify more than one medium by using the Boolean OR operator, allowing the method to choose the best medium among those specified.

pmedium

[out] Pointer to the **STGMEDIUM**structure that indicates the storage medium containing the returned data through its *tymed* member, and the responsibility for releasing the medium through the value of its *punkOuter* member. If *punkForRelease* is NULL, the receiver of the medium is responsible for releasing it; otherwise, *punkForRelease* points to the **IUnknown** on the appropriate object so its **Release** method can be called. The medium must be allocated and filled in by **IDataObject::GetData**.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK

Data was successfully retrieved and placed in the storage medium provided.

DV_E_LINDEX

Invalid value for *lindex*; currently, only -1 is supported.

DV_E_FORMATETC

Invalid value for *pFormatetc*.

DV_E_TYMED

Invalid *tymed* value.

DV_E_DVASPECT

Invalid *dwAspect* value.

OLE_E_NOTRUNNING

Object application is not running.

STG_E_MEDIUMFULL

An error occurred when allocating the medium.

Remarks

A data consumer calls **IDataObject::GetData** to retrieve data from a data object, conveyed through a storage medium (defined through the STGMEDIUM structure).

Notes to Callers

You can specify more than one acceptable **TYMED** medium with the Boolean OR operator. **IDataObject::GetData** must choose from the OR'd values the medium that best represents the data, do the allocation, and indicate responsibility for releasing the medium.

Data transferred across a stream extends from position zero of the stream pointer through to the position immediately before the current stream pointer (that is, the stream pointer position upon exit).

Notes to Implementers

IDataObject::GetData must check all fields in the **FORMATETC** structure. It is important that **IDataObject::GetData** render the requested aspect and, if possible, use the requested medium. If the data object cannot comply with the information specified in the **FORMATETC**, the method should return DV_E_FORMATETC. If an attempt to allocate the medium fails, the method should return STG_E_MEDIUMFULL. It is important to fill in all of the fields in the **STGMEDIUM** structure.

Although the caller can specify more than one medium for returning the data, **IDataObject::GetData** can supply only one medium. If the initial transfer fails with the selected medium, this method can be implemented to try one of the other media specified before returning an error.

See Also

IDataObject::GetDataHere, **IDataObject::SetData**, **FORMATETC**, **STGMEDIUM**

IDataObject::GetDataHere Quick Info

Called by a data consumer to obtain data from a source data object. This method differs from the **GetData** method in that the caller must allocate and free the specified storage medium.

```
HRESULT GetDataHere(
  FORMATETC * pFormatetc,     //Pointer to the FORMATETC structure
  STGMEDIUM * pmedium         //Pointer to the STGMEDIUM structure
);
```

Parameters

pFormatetc

[in] Pointer to the FORMATETC structure that defines the format, medium, and target device to use when passing the data. Only one medium can be specified in **TYMED**, and only the following TYMED values are valid: TYMED_STORAGE, TYMED_STREAM, TYMED_HGLOBAL, or TYMED_FILE.

pmedium

[out] Pointer to the STGMEDIUM structure that defines the storage medium containing the data being transferred. The medium must be allocated by the caller and filled in by **IDataObject::GetDataHere**. The caller must also free the medium. The implementation of this method must always supply a value of NULL for the *punkForRelease* member of the STGMEDIUM structure to which this parameter points.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK

Data was successfully retrieved and placed in the storage medium provided.

DV_E_LINDEX

Invalid value for *lindex*; currently, only -1 is supported.

DV_E_FORMATETC

Invalid value for *pFormatetc*.

DV_E_TYMED

Invalid *tymed* value.

DV_E_DVASPECT

Invalid *dwAspect* value.

OLE_E_NOTRUNNING

Object application is not running.

STG_E_MEDIUMFULL

The medium provided by the caller is not large enough to contain the data.

Remarks

The **IDataObject::GetDataHere** method is similar to **IDataObject::GetData**, except that the caller must both allocate and free the medium specified in *pmedium*. **GetDataHere** renders the data described in a **FORMATETC** structure and copies the data into that caller-provided **STGMEDIUM** structure. For example, if the medium is TYMED_HGLOBAL, this method cannot resize the medium or allocate a new *hGlobal*.

Some media are not appropriate in a call to **GetDataHere**, including GDI types such as metafiles. The **GetDataHere** method cannot put data into a caller-provided metafile. In general, the only storage media it is necessary to support in this method are TYMED_ISTORAGE, TYMED_ISTREAM, and TYMED_FILE.

When the transfer medium is a stream, OLE makes assumptions about where the data is being returned and the position of the stream's seek pointer. In a **GetData** call, the data returned is from stream position zero through just before the current seek pointer of the stream (that is, the position on exit). For **GetDataHere**, the data returned is from the stream position on entry through just before the position on exit.

See Also

IDataObject::GetData, **FORMATETC**, **STGMEDIUM**

IDataObject::QueryGetData Quick Info

Determines whether the data object is capable of rendering the data described in the **FORMATETC** structure. Objects attempting a paste or drop operation can call this method before calling **IDataObject::GetData** to get an indication of whether the operation may be successful.

```
HRESULT QueryGetData(
  FORMATETC * pFormatetc      //Pointer to the FORMATETC structure
);
```

Parameter

pFormatetc

[in] Pointer to the FORMATETC structure defining the format, medium, and target device to use for the query.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK

Subsequent call to **IDataObject::GetData** would probably be successful.

DV_E_LINDEX

Invalid value for *lindex*; currently, only -1 is supported.

DV_E_FORMATETC

Invalid value for *pFormatetc*.

DV_E_TYMED

Invalid *tymed* value.

DV_E_DVASPECT

Invalid *dwAspect* value.

OLE_E_NOTRUNNING

Object application is not running.

Remarks

The client of a data object calls **IDataObject::QueryGetData** to determine whether passing the specified **FORMATETC** structure to a subsequent call to **IDataObject::GetData** is likely to be successful. A successful return from this method does not necessarily ensure the success of the subsequent paste or drop operation.

See Also

IDataObject::GetData, **FORMATETC**

IDataObject::SetData Quick Info

Called by an object containing a data source to transfer data to the object that implements this method.

```
HRESULT SetData(
  FORMATETC * pFormatetc,     //Pointer to the FORMATETC structure
  STGMEDIUM * pmedium,        //Pointer to STGMEDIUM structure
  BOOL fRelease               //Indicates which object owns the storage medium after the call is completed
);
```

Parameters

pFormatetc

[in] Pointer to the FORMATETC structure defining the format used by the data object when interpreting the data contained in the storage medium.

pmedium

[in] Pointer to the STGMEDIUM structure defining the storage medium in which the data is being passed.

fRelease

[in] If TRUE, the data object called, which implements **IDataObject::SetData**, owns the storage medium after the call returns. This means it must free the medium after it has been used by calling the **ReleaseStgMedium** function. If FALSE, the caller retains ownership of the storage medium and the data object called uses the storage medium for the duration of the call only.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK

Data was successfully transferred.

E_NOTIMPL

This method is not implemented for the data object.

DV_E_LINDEX

Invalid value for *lindex*; currently, only -1 is supported.

DV_E_FORMATETC

Invalid value for *pFormatetc*.

DV_E_TYMED

Invalid *tymed* value.

DV_E_DVASPECT

Invalid *dwAspect* value.

OLE_E_NOTRUNNING

Object application is not running.

Remarks

IDataObject::SetData allows another object to attempt to send data to the implementing data object. A data object implements this method if it supports receiving data from another object. If it does not support this, it should be implemented to return E_NOTIMPL.

The caller allocates the storage medium indicated by the *pmedium*, in which the data is passed. The data object called does not take ownership of the data until it has successfully received it and no error code is returned. The value of the *fRelease* parameter indicates the ownership of the medium after the call returns. FALSE indicates the caller still owns the medium, and the data object only has the use of it during the call; TRUE indicates that the data object now owns it and must release it when it is no longer needed.

The type of medium (TYMED) specified in the *pformatetc* and *pmedium* parameters must be the same. For example, one cannot be an hGlobal (global handle) and the other a stream.

See Also

IDataObject::GetData, **FORMATETC**, **STGMEDIUM**

IDropSource

Quick Info

The **IDropSource** interface is one of the interfaces you implement to provide drag-and-drop operations in your application. It contains methods used in any application used as a data source in a drag-and-drop operation. The data source application in a drag-and-drop operation is responsible for:

- Determining the data being dragged based on the user's selection.
- Initiating the drag-and-drop operation based on the user's mouse actions.
- Generating some of the visual feedback during the drag-and-drop operation, such as setting the cursor and highlighting the data selected for the drag-and-drop operation.
- Canceling or completing the drag-and-drop operation based on the user's mouse actions.
- Performing any action on the original data caused by the drop operation, such as deleting the data on a drag move.

IDropSource contains the methods for generating visual feedback to the end user and for canceling or completing the drag-and-drop operation. You also need to call the **DoDragDrop**, **RegisterDragDrop**, and **RevokeDragDrop** functions in drag-and-drop operations.

When to Implement

Implement **IDropSource** if you are developing a container or server application that can act as a data source for a drag-and-drop operation. The **IDropSource** interface is only required during the drag-and-drop operation.

If you implement the **IDropSource** interface, you must also implement the **IDataObject** interface on the same object to represent the data being transferred.

You can use the same implementation of **IDataObject** for drag-and-drop data as for the data object offered to the clipboard. Once you have implemented clipboard operations in your application, you can add drag-and-drop operations with only a little extra work.

When to Use

You don't usually call **IDropSource** methods directly. Instead, your data source calls the **DoDragDrop** function when it detects that the user has initiated a drag-and-drop operation. Then, **DoDragDrop** calls the **IDropSource** methods during the drag-and-drop operation.

For example, **DoDragDrop** calls **IDropSource::GiveFeedback** when you need to change the cursor shape or when you need to provide some other visual feedback. **DoDragDrop** calls **IDropSource::QueryContinueDrag** when there is a change in the mouse button state to determine if the drag-and-drop operation was canceled or completed.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IDropSource Methods	**Description**
QueryContinueDrag	Determines whether a drag-and-drop operation should continue.

GiveFeedback	Gives visual feedback to an end user during a drag-and-drop operation.

See Also

DoDragDrop, IDataObject, IDropTarget

IDropSource::GiveFeedback

Quick Info

Enables a source application to give visual feedback to the end user during a drag-and-drop operation by providing the **DoDragDrop** function with an enumeration value specifying the visual effect.

```
HRESULT GiveFeedback(
  DWORD dwEffect      //Effect of a drop operation
);
```

Parameter

dwEffect

[in] The **DROPEFFECT** value returned by the most recent call to **IDropTarget::DragEnter**, **IDropTarget::DragOver**, or **IDropTarget::DragLeave**. For a list of values, see the **DROPEFFECT** enumeration.

Return Values

This method supports the standard return values E_INVALIDARG, E_UNEXPECTED, and E_OUTOFMEMORY, as well as the following:

S_OK

The method completed its task successfully, using the cursor set by the source application.

DRAGDROP_S_USEDEFAULTCURSORS

Indicates successful completion of the method, and requests OLE to update the cursor using the OLE-provided default cursors.

Remarks

When your application detects that the user has started a drag-and-drop operation, it should call the **DoDragDrop** function. **DoDragDrop** enters a loop, calling **IDropTarget::DragEnter** when the mouse first enters a drop target window, **IDropTarget::DragOver** when the mouse changes its position within the target window, and **IDropTarget::DragLeave** when the mouse leaves the target window.

For every call to either **IDropTarget::DragEnter** or **IDropTarget::DragOver**, **DoDragDrop** calls **IDropSource::GiveFeedback**, passing it the **DROPEFFECT** value returned from the drop target call.

DoDragDrop calls **IDropTarget::DragLeave** when the mouse has left the target window. Then, **DoDragDrop** calls **IDropSource::GiveFeedback** and passes the DROPEFFECT_NONE value in the *dwEffect* parameter.

The *dwEffect* parameter can include DROPEFFECT_SCROLL, indicating that the source should put up the drag-scrolling variation of the appropriate pointer.

OLE defines a recommended set of cursor shapes that your application should use. See the User Interface Guidelines for more information.

Notes to Implementers

This function is called frequently during the **DoDragDrop** loop, so you can gain performance advantages if you optimize your implementation as much as possible.

IDropSource::GiveFeedback is responsible for changing the cursor shape or for changing the highlighted source based on the value of the *dwEffect* parameter. If you are using default cursors, you can return DRAGDROP_S_USEDEFAULTCURSORS, which causes OLE to update the cursor for you, using its defaults.

See Also

DoDragDrop, **IDropTarget**

IDropSource::QueryContinueDrag Quick Info

Determines whether a drag-and-drop operation should be continued, canceled, or completed. You do not call this method directly. The OLE **DoDragDrop** function calls this method during a drag-and-drop operation.

```
HRESULT QueryContinueDrag(
  BOOL fEscapePressed,     //Status of escape key since previous call
  DWORD grfKeyState        //Current state of keyboard modifier keys
);
```

Parameters

fEscapePressed

[in] Specifies whether the Esc key has been pressed since the previous call to **IDropSource::QueryContinueDrag** or to **DoDragDrop** if this is the first call to **QueryContinueDrag**. A TRUE value indicates the end user has pressed the escape key; a FALSE value indicates it has not been pressed.

grfKeyState

[in] Current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags MK_CONTROL, MK_SHIFT, MK_ALT, MK_BUTTON, MK_LBUTTON, MK_MBUTTON, and MK_RBUTTON.

Return Values

This method supports the standard return values E_UNEXPECTED and E_OUTOFMEMORY, as well as the following:

S_OK

The drag operation should continue. This result occurs if no errors are detected, the mouse button starting the drag-and-drop operation has not been released, and the Esc key has not been detected.

DRAGDROP_S_DROP

The drop operation should occur completing the drag operation. This result occurs if *grfKeyState* indicates that the key that started the drag-and-drop operation has been released.

DRAGDROP_S_CANCEL

The drag operation should be canceled with no drop operation occurring. This result occurs if *fEscapePressed* is TRUE, indicating the Esc key has been pressed.

Remarks

The **DoDragDrop** function calls **IDropSource::QueryContinueDrag** whenever it detects a change in the keyboard or mouse button state during a drag-and-drop operation. **IDropSource::QueryContinueDrag** must determine whether the drag-and-drop operation should be continued, canceled, or completed based on the contents of the parameters *grfKeyState* and *fEscapePressed*.

See Also

DoDragDrop

IDropTarget Quick Info

The **IDropTarget** interface is one of the interfaces you implement to provide drag-and-drop operations in your application. It contains methods used in any application that can be a target for data during a drag-and-drop operation. A drop-target application is responsible for:

- Determining the effect of the drop on the target application.
- Incorporating any valid dropped data when the drop occurs.
- Communicating target feedback to the source so the source application can provide appropriate visual feedback such as setting the cursor.
- Implementing drag scrolling.
- Registering and revoking its application windows as drop targets.

The **IDropTarget** interface contains methods that handle all these responsibilities except registering and revoking the application window as a drop target, for which you must call the **RegisterDragDrop** and the **RevokeDragDrop** functions.

When to Implement

Implement the **IDropTarget** interface if you are developing an application that can act as a target for a drag-and-drop operation. The **IDropTarget** interface is associated with your application windows and is implemented on your window objects. Call the **RegisterDragDrop** function to register your window objects as drop targets.

When to Use

You do not call the methods of **IDropTarget** directly. The **DoDragDrop** function calls the **IDropTarget** methods during the drag-and-drop operation.

For example, **DoDragDrop** calls **IDropTarget::DragEnter** when it detects the mouse has moved over a window that is registered as a drag target. Once the mouse has entered a drag-target window, **DoDragDrop** calls **IDropTarget::DragOver** as the mouse moves through the window and calls **IDropTarget::DragLeave** if the mouse leaves the target window or if the user cancels or completes the drag-and-drop operation. **DoDragDrop** calls **IDropTarget::Drop** if the drop finally occurs.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IDropTarget Methods	**Description**
DragEnter	Determines whether a drop can be accepted and its effect if it is accepted.
DragOver	Provides target feedback to the user through the **DoDragDrop** function.
DragLeave	Causes the drop target to suspend its feedback actions.
Drop	Drops the data into the target window.

See Also

DoDragDrop, IDropSource, RegisterDragDrop, RevokeDragDrop

IDropTarget::DragEnter Quick Info

Indicates whether a drop can be accepted, and, if so, the effect of the drop.

```
HRESULT DragEnter(
  IDataObject * pDataObject,    //Pointer to the interface of the source data object'
  DWORD grfKeyState,            //Current state of keyboard modifier keys
  POINTL pt,                    //Pointer to the current cursor coordinates
  DWORD * pdwEffect             //Pointer to the effect of the drag-and-drop operation
);
```

Parameters

pDataObject

[in] Pointer to the **IDataObject** interface on the data object. This data object contains the data being being transferred in the drag-and-drop operation. If the drop occurs, this data object will be incorporated into the target.

grfKeyState

[in] Current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags MK_CONTROL, MK_SHIFT, MK_ALT, MK_BUTTON, MK_LBUTTON, MK_MBUTTON, and MK_RBUTTON.

pt

[in] Pointer to the current cursor coordinates in the coordinate space of the drop-target window.

pdwEffect

[in, out] On entry, pointer to the value of the *pdwEffect* parameter of the **DoDragDrop** function. On return, must contain one of the effect flags from the **DROPEFFECT** enumeration, which indicates what the result of the drop operation would be.

Return Values

This method supports the standard return values E_OUTOFMEMORY, E_INVALIDARG, and E_UNEXPECTED, as well as the following:

S_OK

The method completed its task successfully.

Remarks

You do not call **IDropTarget::DragEnter** directly; instead the **DoDragDrop** function calls it to determine the effect of a drop the first time the user drags the mouse into the registered window of a drop target.

To implement **IDropTarget::DragEnter**, you must determine whether the target can use the data in the source data object by checking three things:

- The format and medium specified by the data object
- The input value of *pdwEffect*
- The state of the modifier keys

To check the format and medium, use the **IDataObject** pointer passed in the *pDataObject* parameter to call **IDataObject::EnumFormatEtc** so you can enumerate the FORMATETC structures the source data object supports. Then call **IDataObject::QueryGetData** to determine whether the data object can render the data on the target by examining the formats and medium specified for the data object.

On entry to **IDropTarget::DragEnter**, the *pdwEffect* parameter is set to the effects given to the *pdwOkEffect* parameter of the **DoDragDrop** function. The **IDropTarget::DragEnter** method must choose one of these effects or disable the drop.

The following modifier keys affect the result of the drop:

Key Combination	User-Visible Feedback	Drop Effect
CTRL + SHIFT	=	DROPEFFECT_LINK
CTRL	+	DROPEFFECT_COPY
No keys or SHIFT	None	DROPEFFECT_MOVE

On return, the method must write the effect, one of the members of the DROPEFFECT enumeration, to the *pdwEffect* parameter. **DoDragDrop** then takes this parameter and writes it to its *pdwEffect* parameter. You communicate the effect of the drop back to the source through **DoDragDrop** in the *pdwEffect* parameter. The **DoDragDrop** function then calls **IDropSource::GiveFeedback** so that the source application can display the appropriate visual feedback to the user through the target window.

See Also

DoDragDrop, **IDropSource**, **IDropTarget**, **RegisterDragDrop**, **RevokeDragDrop**

IDropTarget::DragLeave Quick Info

Removes target feedback and releases the data object.

HRESULT DragLeave(*void***);**

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The method completed its task successfully.

Remarks

You do not call this method directly. The **DoDragDrop** function calls this method in either of the following cases:

- When the user drags the cursor out of a given target window.
- When the user cancels the current drag-and-drop operation.

To implement **IDropTarget::DragLeave**, you must remove any target feedback that is currently displayed. You must also release any references you hold to the data transfer object.

See Also

DoDragDrop, **IDropSource**, **IDropTarget**, **RegisterDragDrop**, **RevokeDragDrop**

IDropTarget::DragOver Quick Info

Provides target feedback to the user and communicates the drop's effect to the **DoDragDrop** function so it can communicate the effect of the drop back to the source.

```
HRESULT DragOver(
  DWORD grfKeyState,      //Current state of keyboard modifier keys
  POINTL pt,              //Pointer to the current cursor coordinates
  DWORD * pdwEffect       //Pointer to the effect of the drag-and-drop operation
);
```

Parameters

grfKeyState

[in] Current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags MK_CONTROL, MK_SHIFT, MK_ALT, MK_BUTTON, MK_LBUTTON, MK_MBUTTON, and MK_RBUTTON.

pt

[in] Pointer to the current cursor coordinates in the coordinate space of the drop-target window.

pdwEffect

[in, out] Pointer to the current effect flag. Valid values are from the enumeration DROPEFFECT.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The method completed its task successfully.

Remarks

You do not call **IDropTarget::DragOver** directly. The **DoDragDrop** function calls this method each time the user moves the mouse across a given target window. **DoDragDrop** exits the loop if the drag-and-drop operation is canceled, if the user drags the mouse out of the target window, or if the drop is completed.

In implementing **IDropTarget::DragOver**, you must provide features similar to those in **IDropTarget::DragEnter**. You must determine the effect of dropping the data on the target by examining the **FORMATETC** defining the data object's formats and medium, along with the state of the modifier keys. The mouse position may also play a role in determining the effect of a drop. The following modifier keys affect the result of the drop:

Key Combination	User-Visible Feedback	Drop Effect
CTRL + SHIFT	=	DROPEFFECT_LINK
CTRL	+	DROPEFFECT_COPY
No keys or SHIFT	None	DROPEFFECT_MOVE

You communicate the effect of the drop back to the source through **DoDragDrop** in *pdwEffect*. The **DoDragDrop** function then calls **IDropSource::GiveFeedback** so the source application can display the appropriate visual feedback to the user.

On entry to **IDropTarget::DragOver**, the *pdwEffect* parameter must be set to the allowed effects passed to the *pdwOkEffect* parameter of the **DoDragDrop** function. The **IDropTarget::DragOver** method must be able to choose one of these effects or disable the drop.

Upon return, *pdwEffect* is set to one of the members of the DROPEFFECT enumeration. This value is then passed to the *pdwEffect* parameter of **DoDragDrop**. Reasonable values are DROPEFFECT_COPY to copy the dragged data to the target, DROPEFFECT_LINK to create a link to the source data, or DROPEFFECT_MOVE to allow the dragged data to be permanently moved from the source application to the target.

You may also wish to provide appropriate visual feedback in the target window. There may be some target feedback already displayed from a previous call to **IDropTarget::DragOver** or from the initial **IDropTarget::DragEnter**. If this feedback is no longer appropriate, you should remove it.

For efficiency reasons, a data object is not passed in **IDropTarget::DragOver**. The data object passed in the most recent call to **IDropTarget::DragEnter** is available and can be used.

When **IDropTarget::DragOver** has completed its operation, the **DoDragDrop** function calls **IDropSource::GiveFeedback** so the source application can display the appropriate visual feedback to the user.

Notes to Implementers

This function is called frequently during the **DoDragDrop** loop so it makes sense to optimize your implementation of the **DragOver** method as much as possible.

See Also

DoDragDrop, **IDropSource**, **IDropTarget**, **RegisterDragDrop**, **RevokeDragDrop**

IDropTarget::Drop Quick Info

Incorporates the source data into the target window, removes target feedback, and releases the data object.

```
HRESULT Drop(
  IDataObject * pDataObject,    //Pointer to the interface for the source data
  DWORD grfKeyState,            //Current state of keyboard modifier keys
  POINTL pt,                    //Pointer to the current cursor coordinates
  DWORD * pdwEffect             //Pointer to the effect of the drag-and-drop operation
);
```

Parameters

pDataObject

[in] Pointer to the **IDataObject** interface on the data object being transferred in the drag-and-drop operation.

grfKeyState

[in] Current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags MK_CONTROL, MK_SHIFT, MK_ALT, MK_BUTTON, MK_LBUTTON, MK_MBUTTON, and MK_RBUTTON.

pt

[in] Pointer to the current cursor coordinates in the coordinate space of the drop target window.

pdwEffect

[in, out] Pointer to the current effect flag. Valid values are from the enumeration **DROPEFFECT**.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The method completed its tasks successfully.

Remarks

You do not call this method directly. The **DoDragDrop** function calls this method when the user completes the drag-and-drop operation.

In implementing **IDropTarget::Drop**, you must incorporate the data object into the target. Use the formats available in **IDataObject**, available through *pDataObject*, along with the current state of the modifier keys to determine how the data is to be incorporated, such as linking or embedding.

In addition to incorporating the data, you must also clean up as you do in the **IDropTarget::DragLeave** method:

- Remove any target feedback that is currently displayed.
- Release any references to the data object.

You also pass the effect of this operation back to the source application through **DoDragDrop**, so the source application can clean up after the drag-and-drop operation is complete:

- Remove any source feedback that is being displayed.
- Make any necessary changes to the data, such as removing the data if the operation was a move.

See Also

DoDragDrop, **IDropSource**, **IDropTarget**, **RegisterDragDrop**, **RevokeDragDrop**

IEnum*XXXX*

To allow you to enumerate the number of items of a given type that an object maintains, OLE provides a set of enumeration interfaces, one for each type of item.

To use these interfaces, the client asks an object that maintains a collection of items to create an enumerator object. The interface on the enumeration object is one of the enumeration interfaces, all of which have a name of the form **IEnum***Item_name*. The only difference between enumeration interfaces is what they enumerate – there must be a separate enumeration interface for each type of item enumerated. All have the same set of methods, and are used in the same way. For example, by repeatedly calling the **Next** method, the client gets successive pointers to each item in the collection.

The following table lists the set of enumeration interfaces that OLE defines, and the items enumerated.

Enumeration Interface Name	Item Enumerated
IEnumFORMATETC	An array of FORMATETC structures.
IEnumMoniker	The components of a moniker, or the monikers in a table.
IEnumOLEVERB	The different verbs available for an object, in order of ascending verb number.
IEnumSTATDATA	An array of STATDATA structures which contain advisory connection information for a data object.
IEnumSTATSTG	An array of STATSTG structures, which contain statistical information about a storage, stream, or LockBytes object
IEnumString	Strings
IEnumUnknown	Enumerates IUnknown interface pointers.
IEnumVARIANT	A collection of variants. It allows clients to enumerate heterogeneous collections of objects and intrinsic types when the clients cannot or do not know the specific type(s) of elements in the collection.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IEnum*XXXX* Methods	Description
Next	Retrieves a specified number of items in the enumeration

	sequence.
Skip	Skips over a specified number of items in the enumeration sequence.
Reset	Resets the enumeration sequence to the beginning.
Clone	Creates another enumerator that contains the same enumeration state as the current one.

IEnum*XXXX*::Clone

Creates another enumerator that contains the same enumeration state as the current one. Using this function, a client can record a particular point in the enumeration sequence, and then return to that point at a later time. The new enumerator supports the same interface as the original one.

```
HRESULT Clone(
  IEnum<ELT_T> ** ppenum      //Indirect pointer to the enumeration interface on the object
);
```

Parameter

ppenum

[out] Indirect pointer to the enumeration interface on the enumeration object. If the method is unsuccessful, this parameter's value is undefined.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED.

IEnum*XXXX*::Next

Retrieves the next *celt* items in the enumeration sequence. If there are fewer than the requested number of elements left in the sequence, it retrieves the remaining elements. The number of elements actually retrieved is returned through *pceltFetched* (unless the caller passed in **NULL** for that parameter).

```
HRESULT Next(
  ULONG celt,              //Number of elements requested
  ELT_T[ ] rgelt,          //Array of the elements
  ULONG * pceltFetched     //Pointer to the number of elements actually supplied
);
```

Parameters

celt

[in] Number of elements being requested.

rgelt

[out] Array of size *celt* (or larger) of the elements of interest. The type of this parameter depends on the item being enumerated.

pceltFetched

[in, out] Pointer to the number of elements actually supplied in *rgelt*. Caller can pass in **NULL** if *celt* is one.

Return Value

S_OK if the number of elements supplied is *celt*; S_FALSE otherwise.

IEnum*XXXX*::Reset

Resets the enumeration sequence to the beginning.

HRESULT Reset(*void***);**

Return Value

S_OK

Remarks

A call to this method, resetting the sequence, does not guarantee that the same set of objects will be enumerated after the reset, because it depends on the collection being enumerated. A static collection is reset to the beginning, but it can be too expensive for some collections, such as files in a directory, to guarantee this condition.

IEnum*XXXX*::Skip

Skips over the next specified number of elements in the enumeration sequence.

```
HRESULT Skip(
  ULONG celt      //Number of elements to be skipped
);
```

Parameter

celt

[in] Number of elements to be skipped.

Return Value

S_OK if the number of elements skipped is *celt*; otherwise S_FALSE.

IEnumConnectionPoints Quick Info

This interface enumerates connection points. Connectable objects support the following features:

- Outgoing interfaces, such as event sets
- The ability to enumerate the IIDs of the outgoing interfaces
- The ability to connect and disconnect sinks to the object for those outgoing IIDs
- The ability to enumerate the connections that exist to a particular outgoing interface

When to Implement

To support connectable objects, you need to provide four related interfaces:

- **IConnectionPointContainer**
- **IEnumConnectionPoints**
- **IConnectionPoint**
- **IEnumConnections**

The **IConnectionPointContainer** interface indicates the existence of the outgoing interfaces. It provides access to an enumerator sub-object with the **IEnumConnectionPoints** interface. It also provides a connection point sub-object with the **IConnectionPoint** interface. The **IConnectionPoint** interface provides access to an enumerator sub-object with the **IEnumConnections** interface.

The connection point is a separate sub-object to avoid circular reference counting problems.

A connectable object can be asked to enumerate its supported connection points through **IConnectionPointContainer::EnumConnectionPoints**. The resulting enumerator returned from this method implements the interface **IEnumConnectionPoints**, through which a client can access all the individual connection point sub-objects supported within the connectable object itself, where each connection point implements **IConnectionPoint**.

When enumerating connections through **IEnumConnectionPoints**, the enumerator is responsible for calling **IUnknown::AddRef**, and the caller is responsible for later calling **IUnknown::Release** when those pointers are no longer needed.

When to Use

Use the **IEnumConnectionPoints** interface to enumerate all the supported connection points for each outgoing IID.

The prototypes of the methods are as follows:

```
HRESULT Next(
  ULONG cConnections ,          //[in]Number of IConnectionPoint values returned in rgpcn array
  IConnectionPoint **rgpcn ,    //[out]Array to receive enumerated connection points
  ULONG *pcFetched              //[out]Pointer to the actual number of connection points returned in rgpcn array
);

HRESULT Skip(
  ULONG cConnections      //[in]Number of elements to skip
```

```
);

HRESULT Reset(void);

HRESULT Clone(
  IEnumConnectionPoints **ppEnum      //[out]Indirect pointer to newly created enumerator
);
```

Remarks

IEnumConnectionPoints::Next

Enumerates the next *cConnections* elements (**IConnectionPoint** pointers) in the enumerator's list, returning them in *rgpcn* along with the actual number of enumerated elements in *pcFetched*. The enumerator calls **IConnectionPoint::AddRef** for each returned pointer in *rgpcn*, and the caller is responsible for calling **IConnectionPoint::Release** through each pointer when those pointers are no longer needed.

E_NOTIMPL is not allowed as a return value. If an error value is returned, no entries in the *rgpcn* array are valid on exit and require no release.

IEnumConnectionPoints::Skip

Instructs the enumerator to skip the next *cConnections* elements in the enumeration so that the next call to **IEnumConnectionPoints::Next** will not return those elements.

IEnumConnectionPoints::Reset

Instructs the enumerator to position itself at the beginning of the list of elements.

There is no guarantee that the same set of elements will be enumerated on each pass through the list, nor will the elements necessarily be enumerated in the same order. The exact behavior depends on the collection being enumerated. It is too expensive for some collections, such as files in a directory, to maintain a specific state.

IEnumConnectionPoints::Clone

Creates another connection point enumerator with the same state as the current enumerator to iterate over the same list. This method makes it possible to record a point in the enumeration sequence in order to return to that point at a later time.

The caller must release this new enumerator separately from the first enumerator.

See Also

IConnectionPoint, **IConnectionPointContainer**, **IEnumConnections**, **IEnum*Xxxx***

IEnumConnections Quick Info

This interface enumerates the current connections for a connectable object. Connectable objects support the following features:

- Outgoing interfaces, such as event sets
- The ability to enumerate the IIDs of the outgoing interfaces
- The ability to connect and disconnect sinks to the object for those outgoing IIDs
- The ability to enumerate the connections that exist to a particular outgoing interface

When to Implement

To support connectable objects, you need to provide four related interfaces:

- **IConnectionPointContainer**
- **IEnumConnectionPoints**
- **IConnectionPoint**
- **IEnumConnections**

The **IConnectionPointContainer** interface indicates the existence of the outgoing interfaces. It provides access to an enumerator sub-object with the **IEnumConnectionPoints** interface. It also provides a connection point sub-object with the **IConnectionPoint** interface. The **IConnectionPoint** interface provides access to an enumerator sub-object with the **IEnumConnections** interface.

The connection point is a separate sub-object to avoid circular reference counting problems.

Any individual connection point can support enumeration of its currently known connections through **IConnectionPoint::EnumConnections**. The enumerator created by this method implements the interface **IEnumConnections** which deals with the type **CONNECTDATA**. Each **CONNECTDATA** structure contains the **IUnknown** of a connected sink and the *dwConnection* that was returned by **IConnectionPoint::Advise** when that sink was connected. When enumerating connections through **IEnumConnections**, the enumerator is responsible for calling **IUnknown::AddRef** through the pointer in each enumerated structure, and the caller is responsible to later call **IUnknown::Release** when those pointers are no longer needed.

When to Use

Use the **IEnumConnectionPoints** interface to enumerate all the supported connection points for each outgoing IID.

The prototypes of the methods are as follows:

```
HRESULT Next(
  ULONG cConnections ,        //[in]Number of CONNECTDATA structures returned in rgpcd
  CONNECTDATA **rgpcd ,       //[out]Array to receive enumerated CONNECTDATA structures
  ULONG *pcFetched            //[out]Pointer to actual number of CONNECTDATA structures
);

HRESULT Skip(
  ULONG cConnections      //[in]Number of elements to skip
);
```

HRESULT Reset(void);

HRESULT Clone(

**IEnumConnections *****ppEnum* //[out]Indirect pointer to newly created enumerator

);

Remarks

IEnumConnections::Next

Enumerates the next *cConnections* elements (i.e., **CONNECTDATA** structures) in the enumerator's list, returning them in *rgpcd* along with the actual number of enumerated elements in *pcFetched*.

The caller is responsible for calling CONNECTDATA.pUnk->Release for each element in the array once this method returns successfully. If *cConnections* is greater than one, the caller must also pass a non-NULL pointer to *pcFetched* to get the number of pointers it has to release.

E_NOTIMPL is not allowed as a return value. If an error value is returned, no entries in the *rgpcd* array are valid on exit and require no release.

IEnumConnections::Skip

Instructs the enumerator to skip the next *cConnections* elements in the enumeration so that the next call to **IEnumConnections::Next** will not return those elements.

IEnumConnections::Reset

Instructs the enumerator to position itself at the beginning of the list of elements.

There is no guarantee that the same set of elements will be enumerated on each pass through the list. It depends on the collection being enumerated. It is too expensive for some collections, such as files in a directory, to maintain a specific state.

IEnumConnections::Clone

Creates another connection point enumerator with the same state as the current enumerator to iterate over the same list. This method makes it possible to record a point in the enumeration sequence in order to return to that point at a later time.

The caller must release this new enumerator separately from the first enumerator.

See Also

CONNECTDATA, **IConnectionPoint**, **IConnectionPointContainer**, **IEnumConnectionPoints**, **IEnum*Xxxx***

IEnumFORMATETC Quick Info

The **IEnumFORMATETC** interface is used to enumerate an array of **FORMATETC** structures. **IEnumFORMATETC** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

IEnumFORMATETC must be implemented by all data objects to support calls to **IDataObject::EnumFormatEtc**, which supplies a pointer to the enumerator's **IEnumFORMATETC** interface. If the data object supports a different set of **FORMATETC** information depending on the direction of the data (whether a call is intended for the **SetData** or **GetData** method of IDataObject), the implementation of **IEnumFORMATETC** must be able to operate on both.

The order of formats enumerated through the **IEnumFORMATETC** object should be the same as the order that the formats would be in when placed on the clipboard. Typically, this order starts with private data formats and ends with presentation formats such as CF_METAFILEPICT.

When to Use

Call the methods of **IEnumFORMATETC** when you need to enumerate the FORMATETC structures defining the formats and media supported by a given data object. This is necessary in most data transfer operations, such as clipboard and drag-and-drop, so the object on the other end of the data transfer can determine whether the appropriate format and media for the data is supported.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt*, **FORMATETC** ***** *rgelt*, **ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumFORMATETC **** *ppenum***)**

See Also

OleRegEnumFormatEtc, **FORMATETC**, **IEnum*XXXX***

IEnumMoniker Quick Info

The **IEnumMoniker** interface is used to enumerate the components of a moniker or to enumerate the monikers in a table of monikers. **IEnumMoniker** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

You need to implement **IEnumMoniker** if you are writing a new type of moniker and your monikers have an internal structure that can be enumerated. Your implementation of **IMoniker::Enum** must return an enumerator that implements **IEnumMoniker** and can enumerate your moniker's components. If your moniker has no structure that can be enumerated, your **IMoniker::Enum** method can simply return a NULL pointer.

When to Use

Call the methods of the **IEnumMoniker** interface if you need to enumerate the components of a composite moniker, or to enumerate the monikers in a table.

OLE defines two interfaces that supply an **IEnumMoniker** interface pointer:

- The **IMoniker::Enum** method gets a pointer to an **IEnumMoniker** implementation that can enumerate forwards or backwards through the components of the moniker. For a description of how two of the system-supplied types of monikers enumerate their components, see **IMoniker - File Moniker Implementation** and **IMoniker - Generic Composite Moniker Implementation**.
- The **IRunningObjectTable::EnumRunning** method returns a pointer to an **IEnumMoniker** implementation that can enumerate the monikers registered in a Running Object Table.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt*, **IMoniker** * *rgelt*, **ULONG** * *pceltFetched*)

HRESULT Skip(ULONG *celt*)

HRESULT Reset(*void*)

HRESULT Clone(IEnumMoniker ** *ppenum*)

See Also

IEnum*XXXX*, **IMoniker::Enum**, **IRunningObjectTable::EnumRunning**

IEnumOleUndoUnits Quick Info

The **IEnumOleUndoUnits** interface enumerates the undo units on the undo or redo stack.

When to Implement

The undo manager implements this method to enumerate a list of undo units on the undo or redo stack.

When to Use

Use the **IEnumOleUndoUnits** interface to enumerate all the top level undo units on the undo or redo stack.

The prototypes of the methods are as follows:

```
HRESULT Next(
  ULONG cUndoUnits ,          //[in]Number of undo units returned in rgpcd
  CONNECTDATA **rgpcd ,       //[out]Array to receive enumerated undo units
  ULONG *pcFetched            //[out]Pointer to actual number of undo units
);

HRESULT Skip(
  ULONG cUndoUnits        //[in]Number of undo units to skip
);

HRESULT Reset(void);

HRESULT Clone(
  IEnumOleUndoUnits**             //[out]Indirect pointer to newly created enumerator
ppEnum
);
```

Remarks

IEnumOleUndoUnits::Next

Enumerates the next specified number of undo units in the enumerator's list, returning them in *rgpcd* along with the actual number of enumerated elements in *pcFetched*.

The caller is responsible for calling the **Release** method for each element in the array once this method returns successfully. If *cUndoUnits* is greater than one, the caller must also pass a non-NULL pointer to *pcFetched* to get the number of pointers it has to release.

E_NOTIMPL is not allowed as a return value. If an error value is returned, no entries in the *rgpcd* array are valid on exit and require no release.

IEnumOleUndoUnits::Skip

Instructs the enumerator to skip the next specified number of elements in the enumerator so that the next call to **IEnumOleUndoUnits::Next** will not return those elements.

IEnumOleUndoUnits::Reset

Instructs the enumerator to position itself at the beginning of the list of elements.

There is no guarantee that the same set of elements will be enumerated on each pass through the list. It depends on the collection being enumerated. It is too expensive for some collections, such as files in a directory, to maintain a specific state.

IEnumOleUndoUnits::Clone

Creates another undo unit enumerator with the same state as the current enumerator to iterate over the same list. This method makes it possible to record a point in the enumeration sequence in order to return to that point at a later time.

The caller must release this new enumerator separately from the first enumerator.

See Also

IOleUndoManager, **IOleUndoUnit**

IEnumOLEVERB Quick Info

The **IEnumOLEVERB** interface enumerates the different verbs available for an object in order of ascending verb number. An enumerator that implements the **IEnumOLEVERB** interface is returned by **IOleObject::EnumVerbs**. **IEnumOLEVERB** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

You typically do not have to implement this interface. The OLE default handler provides an implementation that supplies the entries in the registry. Because calls to **IOleObject::EnumVerb** are always routed through the default handler, an OLE application can let the default handler do the work by implementing **IOleObject::EnumVerb** as a stub that simply returns **OLE_S_USEREG**. This informs the default handler that it should create the enumerator for you.

When to Use

Call this interface if you need to list the verbs than an OLE object supports.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt*, **LPOLEVERB** *rgelt*, **ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumOLEVERB ** *ppenum***)**

See Also

OLEVERB, **IEnum*XXXX***

IEnumSTATDATA Quick Info

The **IEnumSTATDATA** interface is used to enumerate through an array of **STATDATA** structures, which contain advisory connection information for a data object. **IEnumSTATDATA** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

IEnumSTATDATA is implemented to enumerate advisory connections. Most applications will not implement this directly, but will use the OLE-provided implementation. Pointers to this implementation are available in two ways:

- In a data object, call **CreateDataAdviseHolder** to get a pointer to the OLE data advise holder object, and then, to implement **IDataObject::EnumDAdvise**, call **IDataAdviseHolder::EnumAdvise**, which creates the enumeration object and supplies a pointer to the implementation of **IEnumSTATDATA**.
- In a compound document object, call **CreateOLEAdviseHolder** to get a pointer to the OLE advise holder object, and then, to implement **IOleObject::EnumAdvise**, call **IOleAdviseHolder::EnumAdvise**, which creates the enumeration object and supplies a pointer to the implementation of **IEnumSTATDATA**.

When to Use

Containers usually call methods that return a pointer to **IEnumSTATDATA** so the container can use its methods to enumerate the existing advisory connections, and use this information to instruct an object to release each of its advisory connections prior to closing down. **IDataObject::EnumDAdvise**, **IDataAdviseHolder::EnumAdvise**, **IOleAdviseHolder::EnumAdvise**, and **IOleCache::EnumCache** methods all supply a pointer to **IEnumSTATDATA**.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt*, **STATDATA *** *rgelt*, **ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumSTATDATA ** *ppenum***)**

See Also

STATDATA, **IEnum*XXXX***, **IOleCache::EnumCache**, **IDataObject::EnumDAdvise**, **IDataAdviseHolder::EnumAdvise**, **IOleObject::EnumAdvise**

IEnumSTATPROPSETSTG Quick Info

The **IEnumSTATPROPSETSTG** interface is used to iterate through an array of **STATPROPSETSTG** structures, which contain statistical information about the property sets managed by the current instance of **IPropertySetStorage**. **IEnumSTATPROPSETSTG** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to IEnum*XXXX*.

The implementation defines the order in which the property sets are enumerated. Property sets that are present when the enumerator is created, and are not removed during the enumeration, will be enumerated only once. Property sets added or deleted while the enumeration is in progress may or may not be enumerated, but, if enumerated, will not be enumerated more than once.

For information on how the OLE compound document implementation of **IEnumSTATPROPSETSTG:Next** supplies members of the STATPROPSETSTG structure, refer to **IEnumSTATPROPSETSTG--Compound File Implementation**.

When to Implement

IEnumSTATPROPSETSTG is implemented to enumerate the property sets supported by the current property set storage object. If you are using the compound file implementation of the storage object, a pointer to which is available through a call to **StgCreateDocfile**, **IEnumSTATPROPSETSTG** is implemented on that object, and a pointer is returned through a call to **IPropertySetStorage::Enum**. If you are doing a custom implementation of **IPropertySetStorage**, you need to implement **IEnumSTATPROPSETSTG** to fill in a caller-allocated array of **STATPROPSETSTG** structures, each of which contains information about the nested elements in the storage object.

When to Use

Call **IPropertySetStorage::Enum** to return a pointer to **IEnumSTATPROPSETSTG**, the methods of which can then be called to enumerate STATPROSETSTG structures so the application can manage its property sets.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt*, **STATPROPSETSTG *** *rgelt*, **ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumSTATPROPSETSTG ** *ppenum***)**

See Also

IPropertyStorage::Enum

IEnumSTATPROPSETSTG-Compound File Implementation

The compound file implementation of **IEnumSTATPROPSETSTG** interface is used to enumerate through an array of **STATPROPSETSTG** structures, which contain statistical information about the properties managed by the compound file implementation of **IPropertySetStorage**, which is associated with a current compound file storage object.

When to Use

Call the methods of **IEnumSTATPROPSETSTG** to enumerate STATPROPSETSTG structures, each of which provides information about one of the property sets associated with the compound file storage object.

Remarks

IEnumSTATPROPSETSTG::Next

Gets the next one or more **STATPROPSETSTG** structures (the number is specified by the *celt* parameter). The **STATPROPSETSTG** elements provided through a call to the compound file implementation of **IEnumSTATPROPSETSTG::Next** follow these rules:

- If **IEnumSTATPROPSETSTG::Next** cannot provide STATPROPSETSTG.fmtid, zeros are written to that member. This occurs when the property set does not have a predefined name (such as \ 005SummaryInformation) and is not a legal value.
- The DocumentSummaryInformation property set is special, in that it may have two property set sections. This property set is described in the section titled The DocumentSummaryInformation Property Set. The second section is referred to as the User-Defined Properties. Each section is identified with a unique Format ID. When **IPropertySetStorage::Enum** is used to enumerate property sets, the User-Defined Property Set will not be enumerated.

Note If you always create a property set using **IPropertySetStorage::Create**, then, since a "Character GUID" is created for the storage name, **IEnumSTATPROPSETSTG::Next** will return a nonzero, valid format identifier for the property set[STATPROPSETSTG.fmtid].

- The STATPROPSETSTG.grfFlags member does not necessarily reflect whether the property set is ANSI or not. If PROPSETFLAG_ANSI is set, the property set is definitely ANSI. If PROPSETFLAG_ANSI is clear, the property set could be either Unicode or non-Unicode, because it is not possible to tell whether it is ANSI without opening it.
- The STATPROPSETSTG.grfFlags member does reflect whether the property set is simple or not, so the setting of the PROPSETFLAG_NONSIMPLE flag is always valid.
- If **IEnumSTATPROPSETSTG::Next** cannot provide STATPROPSETSTG.clsid, it is set to all zeroes (CLSID_NULL). In the OLE compound file implementation, this occurs when the property set is simple (the PROPSETFLAG_NONSIMPLE flag is not set), or is non-simple but the CLSID was not explicitly set. For non-simple property sets, the CLSID that is received is the one that is maintained by the underlying **IStorage**.
- If **IEnumSTATPROPSETSTG::Next** cannot provide the time fields [*ctime*, *mtime*, *atime*], each non-supported time will be set to zeroes. In the OLE compound file implementation, getting these values depends on retrieving them from the underlying **IStorage** implementation.

IEnumSTATPROPSETSTG::Skip

Skips the number of elements specified in *celt*. Returns S_OK if the specified number of elements are skipped, returns S_FALSE if fewer elements than requested are skipped. In any other case, returns the appropriate error.

IEnumSTATPROPSETSTG::Reset

Sets the cursor to the beginning of the enumeration. If successful, returns S_OK, otherwise, returns STG_E_INVALIDHANDLE.

IEnumSTATPROPSETSTG::Clone

Copies the current enumeration state of this enumerator.

IEnumSTATPROPSTG Quick Info

The **IEnumSTATPROPSTG** interface is used to iterate through an array of **STATPROPSTG** structures, which contain statistical information about an open property storage containing a property set. **IEnumSTATPROPSTG** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to ***IEnumXXXX***.

The implementation defines the order in which the properties in the set are enumerated. Properties that are present when the enumerator is created, and are not removed during the enumeration, will be enumerated only once. Properties added or deleted while the enumeration is in progress may or may not be enumerated, but they will never be enumerated more than once.

Properties with property ID 0 (dictionary), property ID 1 (codepage indicator), or property ID greater than or equal to 0x80000000 are not enumerated.

Enumeration of a non-simple property does not necessarily indicate that the property can be read successfully through a call to **IPropertyStorage::ReadMultiple**. This is because the performance overhead of checking existence of the indirect stream or storage is prohibitive during property enumeration. A client of this interface should code accordingly.

When to Implement

Implement **IEnumSTATPROPSTG** to enumerate the properties within a property set. If you are using the compound file implementation of the storage object, a pointer to which is available through a call to **StgCreateDocfile**, you can then query for a pointer to **IPropertySetStorage**. After calling one of its methods either to open or create a property set, you can get a pointer to **IEnumSTATPROPSTG** through a call to **IPropertyStorage::Enum**. If you are doing a custom implementation of **IPropertyStorage**, you also need to implement **IEnumSTATPROPSTG** to fill in a caller-allocated array of **STATPROPSTG** structures. Each **STATPROPSTG** structure contains information about a simple property.

When to Use

Applications that support storage objects and persistent properties within those objects call **IPropertyStorage::Enum** to return a pointer to **IEnumSTATPROPSTG** to enumerate the properties in the current property set.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt*, **STATPROPSTG *** *rgelt*, **ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumSTATPROPSTG ** *ppenum***)**

See Also

IPropertyStorage::Enum

IEnumSTATPROPSTG-Compound File Implementation

The compound file implementation of the **IEnumSTATPROPSTG** interface is used to enumerate through properties, resulting in **STATPROPSTG** structures, which contain statistical information about the properties managed by the compound file implementation of **IPropertyStorage**, which is associated with a current compound file storage object.

The constructor in the OLE implementation of **IEnumSTATPROPSTG** creates a class that reads the entire property set, and creates a static array which can be shared when **IEnumSTATPROPSTG::Clone** is called.

When to Use

Call the compound file implementation of **IEnumSTATPROPSTG** to enumerate the STATPROPSTG structures that contain information about the properties within the current property set. When you are using the compound file implementation of the property storage interfaces, call **IPropertyStorage::Enum** to return a pointer to **IEnumSTATPROPSTG** to manage the property storage object and the elements within it.

Remarks

IEnumSTATPROPSTG::Next

Gets the next one or more STATPROPSTG structures (the number is specified by the *celt* parameter). Returns S_OK if successful.

IEnumSTATPROPSTG::Skip

Skips the number of elements specified in *celt*. The next element to be enumerated through a call to Next then becomes the element after the ones skipped. Returns S_OK if *celt* elements were skipped; returns S_FALSE if fewer than *celt* elements were skipped.

IEnumSTATPROPSTG::Reset

Sets the cursor to the beginning of the enumeration. If successful, returns S_OK, otherwise, returns STG_E_INVALIDHANDLE.

IEnumSTATPROPSTG::Clone

Uses the constructor for the **IEnumSTATPROPSTG** to create a copy of the array. Because the class that constructs the static array actually contains the object, this function mainly adds to the reference count.

See Also

STATPROPSTG, **IPropertyStorage::Enum**

IEnumSTATSTG Quick Info

The **IEnumSTATSTG** interface is used to enumerate through an array of **STATSTG** structures, which contain statistical information about an open storage, stream, or byte array object. **IEnumSTATSTG** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

IEnumSTATSTG is implemented to enumerate the elements of a storage object. If you are using the compound file implementation of the storage object, a pointer to which is available through a call to **StgCreateDocfile**, **IEnumSTATSTG** is implemented on that object, and a pointer is returned through a call to **IStorage::EnumElements**. If you are doing a custom implementation of a storage object, you need to implement **IEnumSTATSTG** to fill in a caller-allocated array of STATSTG structures, each of which contains information about the nested elements in the storage object.

When to Use

Containers call methods that return a pointer to **IEnumSTATSTG** so the container can manage its storage object and the elements within it. Calls to the **IStorage::EnumElements** method supplies a pointer to **IEnumSTATDATA**. The caller allocates an array of **STATSTG** structures and the **IEnumSTATSTG** methods fill in each structure with the statistics about one of the nested elements in the storage object. If present, the *lpszName* member of the **STATSTG** structure requires additional memory allocations through the **IMalloc** interface, and the caller is responsible for freeing this memory, if it is allocated, by calling the **IMalloc::Free** method. If the *lpszName* member is NULL, no memory is allocated, and, therefore, no memory needs to be freed.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt***, STATSTG *** *rgelt***, ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumSTATSTG ** *ppenum***)**

See Also

CoGetMalloc, **IEnum*XXXX***, **IStorage::EnumElements**, **STATSTG**

IEnumString Quick Info

IEnumString is defined to enumerate strings. LPWSTR is the type that indicates a pointer to a zero-terminated string of wide, i.e., Unicode, characters. **IEnumString** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

It is usually not necessary to implement this interface unless you have use for a custom string enumerator. A system implementation in the bind context object on which is the **IBindCtx** interface also contains an implementation of **IEnumString**. **IBindCtx::EnumObjectParam** returns a pointer to this **IEnumString** interface on an enumerator that can return the keys of the bind context's string-keyed table of pointers.

When to Use

Call the methods of **IEnumString** to enumerate through a set of strings.

The prototypes of the member functions are as follows:

HRESULT Next(ULONG *celt*, **LPOLESTR *** *rgelt*, **ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumString ** *ppenum***)**

IEnumUnknown Quick Info

This enumerator enumerates objects with the **IUnknown** interface. It can be used to enumerate through the objects in a component containing multiple objects. **IEnumUnknown** has the same methods as all enumerator interfaces: **Next**, **Skip**, **Reset**, and **Clone**. For general information on these methods, refer to **IEnum*XXXX***.

When to Implement

You can implement this whenever you want a caller to be able to enumerate the objects contained in another object. You get a pointer to **IEnumUnknown** through a call to **IOleContainer::EnumObjects**.

When to Implement

Call the methods of **IEnumUnknown** to enumerate the objects in a compound document, when you get a pointer to the interface on the enumerator through a call to **IOleContainer::EnumObjects**.

The prototypes of the methods are as follows:

HRESULT Next(ULONG *celt***, IUnknown **** *rgelt*, **ULONG *** *pceltFetched***)**

HRESULT Skip(ULONG *celt***)**

HRESULT Reset(*void***)**

HRESULT Clone(IEnumUnknown ** *ppenum***)**

See Also

IOleContainer

IErrorLog Quick Info

The **IErrorLog** interface is an abstraction for an error log that is used to communicate detailed error information between a client and an object. The caller of the single interface method, **AddError**, simply logs an error where the error is an **EXCEPINFO** structure related to a specific property. The implementer of the interface is responsible for handling the error in whatever way it desires.

IErrorLog is used in the protocol between a client that implements **IPropertyBag** and an object that implements **IPersistPropertyBag**.

When to Implement

A container implements **IErrorLog** to provide a control with a means of logging errors when the control is loading its properties from the container-provided property bag.

When to Use

A control logs calls the single method in this interface to log any errors that occur when it is loading its properties.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IErrorLog Method	Description
AddError	Logs an error, an **EXCEPINFO** structure, in the error log during the property load process for a named property.

See Also

IPersistPropertyBag, IPropertyBag

IErrorLog::AddError Quick Info

Logs an error, an **EXCEPINFO** structure, in the error log during the property load process for a named property.

```
HRESULT AddError(
  LPCOLESTR pszPropName,      //Pointer to the name of the property involved with the error
  LPEXCEPINFO pExcepInfo      //Pointer to the caller-initialized EXCEPINFO structure describing the error
);
```

Parameters

pszPropName

[in] Pointer to the name of the property involved with the error. Cannot be NULL.

pExcepInfo

[in] Pointer to the caller-initialized **EXCEPINFO** structure that describes the error to log. Cannot be NULL.

Return Values

S_OK

The error was logged successfully.

E_FAIL

There was a problem logging the error.

E_OUTOFMEMORY

There was not enough memory to log the error.

E_POINTER

The address in *pszPropName* or *pExceptInfo* is not valid (such as NULL). The caller must supply both.

Remarks

E_NOTIMPL is not a valid return code as the method is the only one in the entire interface.

IExternalConnection Quick Info

The **IExternalConnection** interface enables an embedded object to keep track of external locks on it, thereby enabling the safe and orderly shutdown of the object following silent updates. An object that supports links either to itself or to some portion of itself (a range of cells in a spreadsheet, for example) should implement this interface to prevent possible loss of data during shutdown.

Such data loss can occur when an object happens to have unsaved changes at a time when its stub manager's count of strong external references has reached zero. This situation would arise, for example, at the end of a silent update, when the final link client breaks its connection to the object. With the severing of this connection, the stub manager's count of strong external references would reach zero, causing it to release its pointers to an interface on the object and initiate shutdown of the object. When the object calls **IOleClientSite::SaveObject**, its container's return call to **IPersistStorage::Save** would fail because the stub manager would no longer have a pointer to any interface on the object. Any unsaved changes to the object would be lost.

If the object manages its own count of external locks, rather than relying on the stub manager to do so, it can save its data before the stub manager has a chance to release its pointers. An object can obtain a count of external connections by implementing the **IExternalConnection** interface. The stub manager calls this interface whenever a new strong external reference is added or deleted. The object combines this count with its own tally of strong internal references to maintain an accurate total of all locks.

When to Implement

All embeddable compound-document objects that support links to themselves or portions of themselves should implement **IExternalConnection** to prevent possible data loss during shutdown. In addition, an in-place container should call **OleLockRunning** to hold a strong lock on its embedded objects.

When to Use

An object's stub manager should call **IExternalConnection** whenever an external connection is added or released.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IExternalConnection Methods	Description
AddConnection	Increments count of external locks.
ReleaseConnection	Decrements count of external locks.

IExternalConnection::AddConnection Quick Info

Increments an object's count of its strong external connections (links).

```
HRESULT AddConnection(
    DWORD exconn,          //Type of external connection
    DWORD dwreserved       //Used by OLE to pass connection information
);
```

Parameters

exconn

[in] Type of external connection to the object. The only type of external connection currently supported by this interface is strong, which means that the object must remain alive as long as this external connection exists. Strong external connections are represented by the value EXTCONN_STRONG = 0x0001, which is defined in the enumeration **EXTCONN**.

dwreserved

[in] Passes information about the connection. This parameter is reserved for use by OLE. Its value can be zero, but not necessarily. Therefore, implementations of **AddConnection** should not contain blocks of code whose execution depends on whether a zero value is returned.

Return Value

DWORD value

The number of reference counts on the object; used for debugging purposes only.

Remarks

An object created by a EXE object server relies on its stub manager to call **IExternalConnection::AddConnection** whenever a link client activates and therefore creates an external lock on the object. When the link client breaks the connection, the stub manager calls **IExternalConnection::ReleaseConnection** to release the lock.

Since DLL object applications exist in the same process space as their objects, they do not use RPC (remote procedure calls) and therefore do not have stub managers to keep track of external connections. Therefore, DLL servers that support external links to their objects must implement **IExternalConnection** so link clients can directly call the interface to inform them when connections are added or released.

The following is a typical implementation for the **AddConnection** method:

```
DWORD XX::AddConnection(DWORD extconn, DWORD dwReserved)
    {
        return extconn&EXTCONN_STRONG ? ++m_cStrong : 0;
    }
```

See Also

IExternalConnection::ReleaseConnection, **IRunnableObject::LockRunning**, **OleLockRunning**

IExternalConnection::ReleaseConnection Quick Info

Decrements an object's count of its strong external connections (references).

```
HRESULT ReleaseConnection(
    DWORD extconn,          //Type of external connection
    DWORD dwreserved,       //Used by OLE to pass connection information
    BOOL fLastReleaseCloses //Indicates whether connection is last one or not
);
```

Parameters

extconn

[in] Type of external connection to the object. The only type of external connection currently supported by this interface is strong, which means that the object must remain alive as long as this external connection exists. Strong external connections are represented by the value EXTCONN_STRONG = 0x0001, which is defined in the enumeration **EXTCONN**.

dwreserved

[in] Passes information about the connection. This parameter is reserved for use by OLE. Its value can be zero, but not necessarily. Therefore, implementations of **ReleaseConnection** should not contain blocks of code whose execution depends on whether a zero value is returned.

fLastReleaseCloses

[in] TRUE specifies that if the connection being released is the last external lock on the object, the object should close. FALSE specifies that the object should remain open until closed by the user or another process.

Return Value

DWORD value

The number of connections to the object; used for debugging purposes only.

Remarks

If *fLastReleaseCloses* equals TRUE, calling **ReleaseConnection** causes the object to shut itself down. Calling this method is the only way in which a DLL object, running in the same process space as the container application, will know when to close following a silent update.

See Also

IExternalConnection::AddConnection

IFillLockBytes

The **IFillLockBytes** interface enables downloading code to write data asynchronously to a structured storage byte array. When the downloading code has new data available, it calls **IFillLockBytes::FillAppend** or **IFillLockBytes::FillAt** to write the data to the byte array. An application attempting to access this data, through calls to the **ILockBytes** interface, can do so even as the downloader continues to make calls to **IFillLockBytes**. If the application attempts to access data that has not already been downloaded through a call to **IFillLockBytes**, then **ILockBytes** returns a new error, E_PENDING.

When to Implement

You normally would not implement this interface. A system developer that wants to provide asynchronous storage for a protocol other than http might implement **IFillLockBytes** as part of the transport layer.

When to Use

You normally would not call this interface. Monikers or other downloading code that provide asynchronous storage use this interface to fill the byte array as data becomes available.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IFillLockBytes Methods	Description
FillAppend	Writes a new block of bytes to end of byte array.
FillAt	Writes a new block of bytes to specified location in byte array.
SetFillSize	Sets expected size of byte array.
Terminate	Notifies byte array wrapper of successful or unsuccessful termination of download.

See Also

IConnectionPoint, **IConnectionPointContainer**, **ILockBytes**, **IProgressNotify**, **IStorage**, **IStream**

Also see **IBinding** and **IBindStatusCallback** in the ActiveX™ SDK.

IFillLockBytes::FillAppend

Writes a new block of bytes to the end of a byte array.

```
HRESULT FillAppend(
    void const *pv                  // Data to be appended to byte array
    ULONG cb                        // Number of bytes to be appended
    ULONG *pcbWritten               // Number of bytes that were successfully appended
);
```

Parameters

pv

[in] Points to the data to be appended to the end of an existing byte array.

cb

[in] Size of pv in bytes.

pcbWritten

[out] Number of bytes that were successfully written.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL.

Remarks

The **FillAppend** method is used for sequential downloading, where bytes are written to the end of the byte array in the order in which they are received. This method obtains the current size of the byte array (*i.e.,* lockbytes object) and writes a new block of data to the end of the array. As each block of data becomes available, the downloader calls this method to write it to the byte array. Subsequent calls by the compound file implementation to **ILockBytes::ReadAt** return any available data or return E_PENDING if data is currently unavailable.

See Also

ILockBytes

IFillLockBytes::FillAt

Writes a new block of data to a specified location in the byte array.

```
HRESULT FillAt(
    ULARGE_INTEGER uIOffset     // Offset from beginning of the byte array
    void const *pv              // Data to be written
    ULONG cb                    // Number of bytes to be written
    ULONG *pcbWritten           // Number of bytes that were successfully written
);
```

Parameters

uIOffset

[in] The offset, expressed in number of bytes, from the first element of the byte array.

pv

[in] Points to the data to be written at the location specified by uIOffset.

cb

[in] Size of pv in bytes.

pcbWritten

[out] Number of bytes that were successfully written.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL in addition to the following:

E_NOTIMPL

The byte array does not support **FillAt**.

Remarks

The **FillAt** method is used for nonsequential downloading (for example, http byte range requests). In nonsequential downloading the caller specifies ranges in the byte array where various blocks of data are to be written. Subsequent calls by the compound file implementation to **ILockBytes::ReadAt** are passed by the byte array wrapper object's own implementation of **ILockBytes** to the underlying byte array. This method is not currently implemented and will return E_NOTIMPL.

Note The system-supplied **IFillLockBytes** implementation does not support **FillAt** and returns E_NOTIMPL.

See Also

IFillLockBytes::FillAppend, **IFillLockBytes - Implementation**, **ILockBytes::ReadAt**

IFillLockBytes::SetFillSize

Sets the expected size of the byte array.

```
HRESULT SetFillSize(
    ULARGE_INTEGER uISize        // Size in bytes of a byte array object
);
```

Parameters

uISize

[in] Size in bytes of the byte array object that is to be filled in subsequent calls to **IFillLockBytes::FillAppend**.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL.

Remarks

If **SetFillSize** has not been called, any call to **ILockBytes::ReadAt** that attempts to access data that has not yet been written using **IFillLockBytes::FillAppend** or **IFillLockBytes::FillAt** will return a new error message, E_PENDING. After **SetFillSize** has been called, any call to **ReadAt** that attempts to access data beyond the current size, as set by **SetFillSize**, returns E_FAIL instead of E_PENDING.

See Also

IFillLockBytes::FillAppend, **IFillLockBytes::FillAt**, **ILockBytes::ReadAt**

IFillLockBytes::Terminate

Informs the byte array that the download has been terminated, either successfully or unsuccessfully.

```
HRESULT Terminate(

  BOOL bCanceled                    // Indicates if download was successful
);
```

Parameters

bCanceled

[in] Download is complete. If TRUE, the download was terminated unsuccessfully. If FALSE, the download terminated successfully.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL.

Remarks

After this method has been called, the byte array will no longer return E_PENDING.

IFillLockBytes - Implementation

The system provides an **IFillLockBytes** implementation as part of its existing implementation of Compound Files. Downloading code can instantiate an asynchronous Compound File object by calling **StgOpenAsyncDocFileOnIFillLockBytes** and can instantiate an asynchronous byte array wrapper object on an existing file or byte array by calling either the **StgGetIFillLockBytesOnFile** function or the **StgGetIFillLockBytesOnILockBytes** function.

When to Use

Currently, URL monikers are the only users of OLE's asynchronous storage implementation.

Remarks

FillAppend

Writes a new block of bytes to the end of a byte array.

FillAt

The system implementation does not support this method. Returns E_NOTIMPL.

SetFillSize

Sets size of byte array. Returns E_FAIL from calls to **ILockBytes::ReadAt** that attempts to access data beyond the upper bound specified

Terminate

Informs byte array that a download has been terminated, either successfully or unsuccessfully.

IFont Quick Info

An OLE font object is an object wrapper around a Windows font object. The OLE font object supports a number of read-write properties as well as a set of methods through its **IFont** interface. It supports the same set of properties (but not the methods) through the dispatch interface **IFontDisp**, which is derived from **IDispatch** to provide access to the font's properties through Automation. The system provides a standard implementation of the font object with both interfaces.

The font object also supports the outgoing interface **IPropertyNotifySink** so a client can determine when font properties change. Because the font object supports at least one outgoing interface, it also implements **IConnectionPointContainer** and related interfaces for this purpose.

The font object provides an *hFont* property, which is a Windows font handle that conforms to the other attributes specified for the font. The font object delays realizing this *hFont* object when possible, so consecutively setting two properties on a font won't cause an intermediate font to be realized. In addition, as an optimization, the system-implemented font object maintains a cache of font handles. Two font objects in the same process that have identical properties will return the same font handle. The font object can remove font handles from this cache at will, which introduces special considerations for clients using the **hFont** property. See the description for **IFont::get_hFont** for more details.

The font object also supports **IPersistStream** so it can save and load itself from an instance of **IStream**. An object that uses a font object internally would normally save and load the font as part of the object's own persistence handling.

In addition, the font object supports **IDataObject**, which can render a property set containing the font's attributes, allowing a client to save these properties as text.

When to Implement

Typically, you use the OLE-provided font object, which implements the **IFont** interface as its primary interface. It allows the caller to manage font properties and to use that font in graphical rendering. Each property in the **IFont** interface includes a get_*PropertyName* method if the property supports read access and a put_*PropertyName* method if the property supports write access. Most of the properties support both read and write access, and thus expose both "get" and "put" methods for these properties.

Property	Type	Access Allowed	Description
Name	BSTR	RW	The name of the font family, e.g. Arial.
Size	CY	RW	The point size of the font, expressed in a **CY** type to allow for fractional point sizes.
Bold	BOOL	RW	Indicates whether the font is boldfaced.
Italic	BOOL	RW	Indicates whether the font is italicized.
Underline	BOOL	RW	Indicates whether the font is underlined.
Strikethrough	BOOL	RW	Indicates whether the font is strikethrough.
Weight	short	RW	The boldness or weight of the font.
Charset	short	RW	The character set used in the font, such as ANSI_CHARSET,

			DEFAULT_CHARSET, or SYMBOL_CHARSET.
hFont	HFONT	R	The Windows font handle that can be selected into a device context for rendering.

When to Use

Use this interface to change or retrieve the properties of a font object.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IFont Methods	Description
get_Name	Gets the name of the font family.
put_Name	Sets a new name for the font family.
get_Size	Gets the point size for the font.
put_Size	Sets the point size for the font.
get_Bold	Indicates whether the font is bold or not.
put_Bold	Sets the boldness property for the font.
get_Italic	Indicates whether the font is italic or not.
put_Italic	Sets the italic property for the font.
get_Underline	Indicates whether the font is underlined or not.
put_Underline	Sets the underline property for the font.
get_Strikethrough	Indicates whether the font is strikethrough or not.
put_Strikethrough	Sets the strikethrough property for the font.
get_Weight	Gets the weight (boldness) for the font.
put_Weight	Sets the weight (boldness) for the font.
get_Charset	Gets the font's character set.
put_Charset	Sets the font's character set.
get_hFont	Returns a Windows **HFONT** handle for the font described by this font object.
Clone	Creates a duplicate font object with a state identical to the current font.
IsEqual	Compares this font object to another for equality.
SetRatio	Converts the scaling factor for this font between logical units and HIMETRIC units (in which is expressed the point size in the Size property).
QueryTextMetrics	Fills a **TEXTMETRIC** structure describing the font.
AddRefHfont	Notifies the font object that the previously

	realized font identified with **hFont** (from **IFont::GetHfont**) should remain valid until **IFont::ReleaseHfont** is called or the font object itself is released.
ReleaseHfont	Notifies the font object that the caller that previously locked this font in the cache with **IFont::AddRefHfont** no longer requires the lock.
SetHdc	Provides a device context handle to the font that describes the logical mapping mode.

See Also

IFont - Ole Implementation, **IFontDisp**

IFont::AddRefHfont Quick Info

Notifies the font object that the previously realized font identified with **hFont** (obtained from **IFont::get_hFont**) should remain valid until **IFont::ReleaseHfont** is called or the font object itself is released completely.

```
HRESULT AddRefHfont(
  HFONT hfont      //Font handle returned from IFont::GetHfont
);
```

Parameters

hfont

[in] Font handle previously realized through **IFont::GetHfont** to be locked in the font object's cache.

Return Values

The method supports the standard return values E_UNEXPECTED and E_INVALIDARG, as well as the following:

S_OK

The font was successfully locked in the cache.

See Also

IFont::get hFont, **IFont:ReleaseHfont**

IFont::Clone Quick Info

Creates a duplicate font object with a state identical to the current font.

```
HRESULT Clone(
  IFont** ppfont      //Indirect pointer to the new font object
);
```

Parameters

ppfont

[out] Indirect pointer to the **IFont** interface on the new font object. The caller must call **IFont::Release** when this new font object is no longer needed.

Return Values

The method supports the standard return values E_UNEXPECTED and E_OUTOFMEMORY, as well as the following:

S_OK

The new font object was successfully created.

E_NOTIMPL

This font object does not support cloning.

E_POINTER

The address in *ppfont* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

The new font object is entirely independent of the first. The caller is responsible for releasing this new object when it is no longer needed. This method does not affect the reference count of the font being cloned.

IFont::get_Bold Quick Info

Indicates whether the font is bold or not.

```
HRESULT get_Bold(
  BOOL* pbold      //Pointer to the font's bold format
);
```

Parameters

pbold

[out] Pointer to a caller-allocated **BOOL** variable that indicates whether the font is bold or not.

Return Values

S_OK

The state was retrieved successfully. If the state is indeterminate, a font object should set **pbold* to FALSE and return S_OK.

E_POINTER

The address in *pbold* is not valid. For example, it may be NULL.

See Also

IFont::put_Bold

IFont::get_Charset Quick Info

Returns the character set used in the font. The character set can be any of those defined for Windows fonts.

```
HRESULT get_Charset(
  short* pcharset      //Pointer to the character set
);
```

Parameters

pcharset

[out] Pointer to the caller-allocated **short** variable that receives the character set value.

Return Values

S_OK

The character set was retrieved successfully.

E_POINTER

The address in *pcharset* is not valid. For example, it may be NULL.

See Also

IFont::put_Charset

IFont::get_hFont Quick Info

Returns a Windows **HFONT** handle for the font described by this font object.

```
HRESULT get_hFont(
  HFONT* phfont      //Pointer to the font handle
);
```

Parameters

phfont

[out] Pointer to the caller-allocated **HFONT** variable that receives the font handle. The caller does not own this resource and must not attempt to destroy the font.

Return Values

The method supports the standard return values E_UNEXPECTED and E_OUTOFMEMORY, as well as the following:

S_OK

The font handle was retrieved successfully.

E_POINTER

The address in *phfont* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

The font object maintains ownership of the **HFONT** and can destroy it at any time without prior notification. If the caller needs to secure this font for a limited period of time, it can call **IFont::AddRefHfont** and **IFont::ReleaseHfont**.

See Also

IFont::AddRefHfont, **IFont::ReleaseHfont**

IFont::get_Italic Quick Info

Indicates whether the font is italic or not.

```
HRESULT get_Italic(
  BOOL* pitalic      //Pointer to the font's italic format
);
```

Parameters

pitalic

[out] Pointer to the caller-allocated **BOOL** variable that indicates whether the font is italic.

Return Values

S_OK

The state was retrieved successfully. If the state is indeterminate, a font object should set **pitalic* to FALSE and return S_OK.

E_POINTER

The address in *pitalic* is not valid. For example, it may be NULL.

See Also

IFont::put_Italic

IFont::get_Name Quick Info

Returns a copy of the name of the font family.

```
HRESULT get_Name(
  BSTR* pname      //Pointer to the name of the font family
);
```

Parameters

pname

[out] Pointer to the caller-allocated variable that receives the copy of the name. This name must be freed with **SysFreeString** when it is no longer needed.

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The name was returned successfully.

E_POINTER

The address in *pname* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

The caller is responsible for freeing the memory allocated for the name with **SysFreeString**.

See Also

IFont::put_Name

IFont::get_Size

Quick Info

Retrieves the point size of the font expressed in a 64-bit **CY** variable. The upper 32-bits of this value contains the integer point size and the lower 32-bits contains the fractional point size.

```
HRESULT get_Size(
  CY* psize      //Pointer to the font size.
);
```

Parameters

psize

[out] Pointer to the caller-allocated variable that receives the size.

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The size was retrieved successfully.

E_POINTER

The address in *psize* is not valid. For example, it may be NULL.

See Also

IFont::put_Size

IFont::get_Strikethrough Quick Info

Indicates whether the font has the strikethrough property or not.

```
HRESULT get_Strikethrough(
  BOOL* pstrikethrough      //Pointer to the current strikethrough property for the font
);
```

Parameters

pstrikethrough

[out] Pointer to the caller-allocated **BOOL** variable that indicates the current strikethrough property for the font.

Return Values

S_OK

The state was retrieved successfully. If the state is indeterminate, a font object should set **pstrikethrough* to FALSE and return S_OK.

E_POINTER

The address in *pstrikethrough* is not valid. For example, it may be NULL.

See Also

IFont::put_Strikethrough

IFont::get_Underline Quick Info

Indicates whether the font is underlined or not.

```
HRESULT get_Underline(
  BOOL* punderline      //Pointer to the font's underlined property
);
```

Parameters

punderline

[out] Pointer to the caller-allocated **BOOL** variable that indicates whether the font is underlined.

Return Values

S_OK

The state was retrieved successfully. If the state is indeterminate, a font object should set **punderline* to FALSE and return S_OK.

E_POINTER

The address in *punderline* is not valid. For example, it may be NULL.

See Also

IFont::put_Underline

IFont::get_Weight Quick Info

Returns the font weight, where a weight is defined as any of the FW_* values that are valid for Windows fonts.

```
HRESULT get_Weight(
  short* pweight      //Pointer to the font weight
);
```

Parameters

pweight

[out] Pointer to the caller-allocated **short** variable that receives the current font weight.

Return Values

S_OK

The weight was retrieved successfully. If the weight is indeterminate, a font object should store FW_NORMAL in **pweight* and return S_OK.

E_POINTER

The address in *pweight* is not valid. For example, it may be NULL.

See Also

IFont::put_Weight

IFont::IsEqual Quick Info

Compares this font object to another for equivalence.

```
HRESULT IsEqual(
  IFont* pFontOther      //Pointer to the font to compare to this font
);
```

Parameters

pFontOther

[in] Pointer to the **IFont** interface on the font object to be compared to this font. The reference count of the object referred to by this pointer is not affected by the comparison operation.

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The two fonts are equivalent.

S_FALSE

The two fonts are not equivalent.

E_POINTER

The address in *pFontOther* is not valid. For example, it may be NULL.

IFont::put_Bold

Quick Info

Sets the font's current bold property. Changing the Bold property may also change the Weight property. Setting Bold to TRUE sets the Weight to FW_BOLD (700); setting Bold to FALSE sets Weight to FW_NORMAL (400).

```
HRESULT put_Bold(
  BOOL bold      //Bold property for the font
);
```

Parameters

bold

[in] New bold property for the font.

Return Values

S_OK

The bold state was changed successfully.

S_FALSE

The font does not support a bold state. Note that this is not an error condition.

See Also

IFont::get_Bold

IFont::put_Charset

Quick Info

Sets the font's character set.

```
HRESULT put_Charset(
  short charset      //Character set
);
```

Parameters

charset

[in] New character set for the font.

Return Values

The method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The character set was changed successfully.

See Also

IFont::get_Charset

IFont::put_Italic Quick Info

Sets the font's current italic property.

```
HRESULT put_Italic(
  BOOL italic      //Italic property for the font
);
```

Parameters

italic

[in] New italic property for the font.

Return Values

S_OK

The italic state was changed successfully.

S_FALSE

The font does not support an italic state. This value is not an error condition.

See Also

IFont::get_Italic

IFont::put_Name Quick Info

Specifies a new name for the font family.

```
HRESULT put_Name(
  BSTR name      //Name of the font family
);
```

Parameters

name

[in] New name of the font family. This **BSTR** value is both allocated and freed by the caller.

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The name was changed successfully.

E_POINTER

The address in *name* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

The string value is caller allocated and the caller is responsible for freeing it after this call returns.

See Also

IFont::get_Name

IFont::put_Size Quick Info

Sets the current point size of the font given a **CY** structure.

```
HRESULT put_Size(
  CY size      //Size of the font
);
```

Parameters

size

[in] New size of the font.

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The font was resized successfully.

E_POINTER

The value of the *size* parameter is not valid. For example, it does not contain a usable font size.

See Also

IFont::get_Size

IFont::put_Strikethrough Quick Info

Sets the font's current strikethrough property.

```
HRESULT put_Strikethrough(
  BOOL strikethrough      //Strikethrough property for the font
);
```

Parameters

strikethrough

[in] New strikethrough property for the font.

Return Values

S_OK

The strikethrough state was changed successfully.

S_FALSE

The font does not support a strikethrough state. This value is not an error condition.

See Also

IFont::get_Strikethrough

IFont::put_Underline Quick Info

Sets the font's current underline property.

```
HRESULT put_Underline(
  BOOL underline      //Underline property for the font
);
```

Parameters

underline

[in] New underline property for the font.

Return Values

S_OK

The underline state was changed successfully.

S_FALSE

The font does not support an underline state. This value is not an error condition.

See Also

IFont::get_Underline

IFont::put_Weight Quick Info

Sets the font's weight. This property may affect the bold property as well. Bold is set to TRUE if the weight is greater than the average of FW_NORMAL (400) and FW_BOLD (700), that is 550.

```
HRESULT put_Weight(
  short weight      //Weight for the font
);
```

Parameters

weight

[in] New weight for the font.

Return Values

S_OK

The weight was changed successfully.

S_FALSE

This font does not support different weights. This value is not an error condition.

See Also

IFont::get_Weight

IFont::QueryTextMetrics Quick Info

Fills a caller-allocated **TEXTMETRIC** structure for the font. The **TEXTMETRICOLE** structure is defined as a **TEXTMETRICW** structure on Win32 platforms. For more information on this structure, consult the *Win32 Programmer's Reference*.

```
HRESULT QueryTextMetrics(
  TEXTMETRICOLE* ptm      //Pointer to font information structure to be filled
);
```

Parameters

ptm

[out] Pointer to the caller-allocated structure that receives the font information.

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The text metrics were returned successfully.

E_POINTER

The address in *ptm* is not valid. For example, it may be NULL.

Remarks

Notes to Implementers

E_NOTIMPL is not a valid return value. Font objects must always provide their font information through this call unless other errors occur.

IFont::ReleaseHfont Quick Info

Notifies the font object that the caller that previously locked this font in the cache with **IFont::AddRefHfont** no longer requires the lock.

```
HRESULT ReleaseHfont(
    HFONT hfont      //Font handle returned from IFont::GetHfont
);
```

Parameters

hfont

[in] Font handle previously realized through **IFont::GetHfont**. This value was passed to a previous call to **IFont::AddRefHfont** to lock the font, and the caller would now like to unlock the font in the cache.

Return Values

The method supports the standard return values E_UNEXPECTED and E_INVALIDARG, as well as the following:

S_OK

The font was unlocked successfully.

S_FALSE

The font was not locked in the cache. This return value is a benign notification to the caller that it may have a font reference counting problem.

See Also

IFont::AddRefHfont

IFont::SetHdc Quick Info

Provides a device context to the font that describes the logical mapping mode.

```
HRESULT SetHdc(
  HDC hdc      //Device context handle
);
```

Parameters

hdc

[in] Handle to the device context in which to select the font.

Return Values

The method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The font was selected successfully.

E_NOTIMPL

The font selection is not supported through this font object.

Remarks

The logical mapping mode affects the font's internal computation of its point size so that when the caller asks for a font handle by calling **IFont::GetHfont**, the font is already properly scaled to the device context.

Notes to Callers

The caller retains ownership of this device context which must remain valid for the lifetime of the font object. Thus, the device context passed should be a memory device context (from the Win32 function **CreateCompatibleDC**) and not a screen device context (from **CreateDC**, **GetDC**, or **BeginPaint**) because screen device contexts are a limited system resource.

IFont::SetRatio Quick Info

Converts the scaling factor for this font between logical units and HIMETRIC units. HIMETRIC units are used to express the point size in the **IFont::get_Size** and **IFont::put_Size** methods. The values passed to **IFont::SetRatio** are used to compute the display size of the font in logical units from the value in the Size property:

Display Size= (cyLogical/cyHimetric) * Size

```
HRESULT SetRatio(
  long cyLogical ,      //Font size in logical units
  long cyHimetric       //Font size in HIMETRIC units
);
```

Parameters

cyLogical

[in] Font size in logical units.

cyHimetric

[in] Font size in HIMETRIC units.

Return Values

The method supports the standard return values E_UNEXPECTED and E_INVALIDARG, as well as the following:

S_OK

The ratio was set successfully.

See Also

IFont::get_Size, **IFont::put_Size**

IFont - Ole Implementation

The system provides a standard implementation of a font object with the **IFont** interface on top of the underlying system font support. A font object is created through the **OleCreateFontIndirect** function. A font object supports a number of read-write properties, as well as a set of methods, through its **IFont** interface and supports the same set of properties (but not the methods) through a dispatch interface **IFontDisp** which is derived from **IDispatch** to provide access to the font's properties through Automation. The system implementation of the font object supplies both interfaces.

Remarks

The OLE-provided font object implements the complete semantics of the **IFont** and **IFontDisp** interfaces.

See Also

IFont

IFontDisp Quick Info

This interface exposes a font object's properties through Automation. It provides a subset of the **IFont** methods.

When to Implement

A font object implements this interface along with **IFont** to provide access to the font's properties through Automation. Typically, it is not necessary to implement this interface on your own object, since there is an OLE-provided implementation of the font object.

The following table describes the dispIDs for the various font properties.

Symbol	Value
DISPID_FONT_NAME	0
DISPID_FONT_SIZE	2
DISPID_FONT_BOLD	3
DISPID_FONT_ITALIC	4
DISPID_FONT_UNDER	5
DISPID_FONT_STRIKE	6
DISPID_FONT_WEIGHT	7
DISPID_FONT_CHARSET	8

Each property in the **IFontDisp** interface includes a get_*PropertyName* method if the property supports read access and a put_*PropertyName* method if the property supports write access. Most of the properties support both read and write access and thus expose both "get" and "put" methods for these properties.

Property	Type	Access Allowed	Description
Name	BSTR	RW	The facename of the font, e.g. Arial.
Size	CY	RW	The point size of the font, expressed in a **CY** type to allow for fractional point sizes.
Bold	BOOL	RW	Indicates whether the font is boldfaced.
Italic	BOOL	RW	Indicates whether the font is italicized.
Underline	BOOL	RW	Indicates whether the font is underlined.
Strikethrough	BOOL	RW	Indicates whether the font is strikethrough.
Weight	short	RW	The boldness of the font.
Charset	short	RW	The character set used in the font, such as ANSI_CHARSET, DEFAULT_CHARSET, or SYMBOL_CHARSET.

When to Use

Use this interface to change or retrieve the properties of a font object through the **IDispatch::Invoke** method in Automation.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

See Also

IFont

IFontDisp - Ole Implementation

The system provides a standard implementation of a font object with the **IFontDisp** interface on top of the underlying system font support. A font object is created through the function **OleCreateFontIndirect**. A font object supports a number of read-write properties as well as a set of methods through its interface **IFont** and supports the same set of properties (but not the methods) through a dispatch interface **IFontDisp** which is derived from **IDispatch** to provide access to the font's properties through Automation. The system implementation of the font object supplies both interfaces.

See Also

IFont, **IFontDisp**

ILayoutStorage

The **ILayoutStorage** interface enables an application to optimize the layout of its compound files for efficient downloading across a slow link. The goal is to enable a browser or other application to download data in the order in which it will actually be needed.

To optimize a compound file an application first saves it, then calls **StgOpenLayoutDocfile** to reopen the file using a special Compound Files implementation that exposes **ILayoutStorage** on its root storage. Once the storage is open, the application queries for a pointer to **ILayoutStorage**.

Using this pointer, the application can either provide explicit layout instructions or obtain them by monitoring the pattern of access to the compound file as it is loaded into memory.

- To monitor the order in which a compound file's data is actually accessed, the application calls **ILayoutStorage::BeginMonitor**, after which the Compound Files implementation assumes that any operation performed on the storage or stream object is part of the desired access pattern. When the specified compound file is completely loaded, the application halts monitoring by calling **ILayoutStorage::EndMonitor**.

However the desired layout information is obtained, the application calls **ILayoutStorage::ReLayoutDocfile** to rewrite the compound file using either the explicit layout instructions or the monitored layout access patterns.

When to Implement

You do not need to implement this interface. A special implementation of Compound Files exposes **ILayoutStorage** on the root storage object.

When to Use

Use **ILayoutStorage** to optimize your application's compound files. The interface is exposed in dflayout.dll, which is included with the Win32 SDK. If you are a web master, you can optimize compound files exposed in your web site by using the **Docfile Layout Tool** (dflayout.exe), which is also included with the Win32 SDK.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

ILayout Storage Method	Description
LayoutScript	Provides explicit layout instructions.
BeginMonitor	Monitors data access to a file.
EndMonitor	Ends monitoring of data access.
ReLayoutDocfile	Rewrites file using layout information.

See Also

ILayoutStorage

ILayoutStorage::LayoutScript

Provides explicit directions for reordering the storages, streams, and controls in a compound file to match the order in which they are accessed during the download.

```
HRESULT LayoutScript(
StorageLayout *pStorageLayout      // Pointer to first element in an array of structures.
DWORD nEntries                     // Number of elements in the array
DWORD glfInterleavedFlag           // Reserved for future use
);
```

Parameters

pStorageLayout

[in] Pointer to an array of **StorageLayout** structures.

nEntries

[in] Number of entries in the array of StorageLayout structures.

glfInterleavedFlag

[in] Reserved for future use.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as the following:

STG_E_INVALIDPOINTER

The storage layout pointer is invalid.

STG_E_INVALIDFLAG

The value of *glfInterleavedFlag* is invalid.

STG_E_PATHNOTFOUND

The new docfile name specified is invalid.

STG_E_INSUFFICIENTMEMORY

There is insufficient memory to complete the operation.

STG_E_INVALIDPARAMETER

One of the parameters is invalid.

STG_E_INUSE

BeginMonitor was called while **ILayoutStorage** was already monitoring.

Remarks

To provide explicit layout instructions, the application calls **ILayoutStorage::LayoutScript**, passing an array of **StorageLayout** structures. Each structure defines a single storage or stream data block and specifies where the block is to be written in the **ILockBytes** byte array.

An application can combine scripted layout with monitoring, as the structure of a particular compound file may dictate.

When the optimal data-layout pattern of an entire compound file has been determined, the application calls **ILayoutStorage::ReLayoutDocfile** to restructure the compound file to match the order in which its data sectors were accessed.

See Also

ILayoutStorage::ReLayoutDocfile, **ILockBytes**, **StorageLayout**

ILayoutStorage::BeginMonitor

Allows monitoring of a loading operation to begin. When the operation is complete, the application must call **ILayoutStorage::EndMonitor**.

HRESULT BeginMonitor(void**);**

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as the following:

STG_E_INUSE

BeginMonitor was called while **ILayoutStorage** was already monitoring.

Remarks

Normally an application calls **BeginMonitor** before the actual loading begins. Once this method has been called, the compound file implementation regards any operation performed on the files storages and streams as part of the desired access pattern. The result is a layout script like that created explicitly by calling **ILayoutStorage::LayoutScript**.

Applications will usually use monitoring to obtain the access pattern of embedded objects. Monitoring also makes possible generic layout tools, such as the Docfile Layout Tool (dflayout.exe) included in the Win32 SDK, that launch existing applications and monitor their access patterns.

A call to **ILayoutStorage::EndMonitor** ends monitoring. Multiple calls to **BeginMonitor**/**EndMonitor** are permitted. Monitoring can also be mixed with calls to **ILayoutStorage::LayoutScript**.

See Also

ILayoutStorage::EndMonitor, **ILayoutStorage::LayoutScript**

ILayoutStorage::EndMonitor

Ends monitoring of a compound file. Must be preceded by a call to **ILayoutStorage::BeginMonitor**.

HRESULT EndMonitor(void**);**

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as all return values for **CloseHandle**.

Remarks

A call to **EndMonitor** is generally followed by a call to **ILayoutStorage::RelayoutDocfile**, which uses the access pattern detected by the monitoring to restructure the compound file.

See Also

ILayoutStorage::BeginMonitor, **ILayoutStorage::ReLayoutDocfile**

ILayoutStorage::ReLayoutDocfile

Rewrites the compound file, using the layout script obtained through monitoring, or provided through explicit layout scripting, to create a new compound file.

```
HRESULT ReLayoutDocfile(
OLECHAR *pwcsNewDfName          // Pointer to name of compound file to be rewritten.
);
```

Parameters

pwcsNewDfName

[in] Pointer to the name of the compound file to be rewritten. This name must be a valid filename, distinct from the name of the original compound file. The original compound file will be optimized and written to the new *pwcsNewDfName*.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as the following:

STG_E_INVALIDNAME

The name passed to this function is not a valid filename.

STG_E_UNKNOWN

The layout information has been corrupted and cannot be processed.

ILockBytes

Quick Info

The **ILockBytes** interface is implemented on a byte array object that is backed by some physical storage, such as a disk file, global memory, or a database. It is used by an OLE compound file storage object to give its root storage access to the physical device, while isolating the root storage from the details of accessing the physical storage.

When to Implement

Most applications will not implement the **ILockBytes** interface because OLE provides implementations for the two most common situations:

File-based implementation – If you call **StgCreateDocfile** function to create a compound file storage object, it contains an implementation of **ILockBytes** that is associated with a byte array stored in a physical disk file. The compound file storage object calls the **ILockBytes** methods–you do not call them directly in this implementation.

Memory-based implementation – OLE also provides a byte array object based on global memory that supports an implementation of **ILockBytes**. You can get a pointer through a call to the **CreateILockBytesOnHGlobal** function). Then, to create a compound file storage object on top of that byte array object, call the **StgCreateDocfileOnILockBytes** function. The compound file storage object calls the **ILockBytes** methods – you do not call them directly in this implementation.

There are situations in which it would be useful for an application to provide its own **ILockBytes** implementation. For example, a database application could implement **ILockBytes** to create a byte array object backed by the storage of its relational tables. However, it is strongly recommended that you use the OLE-provided implementations. For a discussion of the advantages of using the OLE implementations rather than creating your own, see the **StgCreateDocfileOnILockBytes** API function, which creates a compound file storage object on top of a caller-provided byte array object.

If you choose to implement your own **ILockBytes** interface, you should consider providing custom marshaling by implementing the **IMarshal** interface as part of your byte array object. The reason for this is that when the OLE-provided implementations of **IStorage** and **IStream** are marshaled to another process, their **ILockBytes** interface pointers are also marshaled to the other process. The default marshaling mechanism creates a proxy byte array object (on which is the **ILockBytes** interface) that transmits method calls back to the original byte array object. Custom marshaling can improve efficiency by creating a remote byte array object that can access the byte array directly.

When to Use

The **ILockBytes** methods are called by the OLE implementations of **IStorage** and **IStream** on the compound file object. Unless you are implementing **IStorage** and **IStream**, you would not need to call **ILockBytes** methods directly. If you write your own **ILockBytes** implementation, you can use the **StgCreateDocfileOnILockBytes** function to create a compound file storage object backed by your implementation of **ILockBytes**.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

ILockBytes Methods	**Description**
ReadAt	Reads a specified number of bytes starting

	at a specified offset from the beginning of the byte array.
WriteAt	Writes a specified number of bytes to a specified location in the byte array.
Flush	Ensures that any internal buffers maintained by the byte array object are written out to the backing storage.
SetSize	Changes the size of the byte array.
LockRegion	Restricts access to a specified range of bytes in the byte array.
UnlockRegion	Removes the access restriction on a range of bytes previously restricted with **ILockBytes::LockRegion**.
Stat	Retrieves a **STATSTG** structure for this byte array object.

ILockBytes::Flush Quick Info

Ensures that any internal buffers maintained by the **ILockBytes** implementation are written out to the underlying physical storage.

HRESULT Flush(*void***);**

Return Values

S_OK

The flush operation was successful.

STG_E_ACCESSDENIED

The caller does not have permission to access the byte array.

STG_E_MEDIUMFULL

The flush operation is not completed because there is no space left on the storage device.

E_FAIL

General failure writing data.

STG_E_TOOMANYFILESOPEN

Under certain circumstances, **Flush** does a dump-and-close to flush, which can lead to a return value of STG_E_TOOMANYFILESOPEN if no file handles are available.

STG_E_INVALIDHANDLE

An underlying file has been prematurely closed, or the correct floppy disk has been replaced by an invalid one.

Remarks

ILockBytes::Flush flushes internal buffers to the underlying storage device.

The OLE-provided implementation of compound files calls this method during a transacted commit operation to provide a two-phase commit process that protects against loss of data.

See Also

IStorage::Commit, **ILockBytes—File-Based Implementation**, **ILockBytes—Global Memory Implementation**

ILockBytes::LockRegion Quick Info

Restricts access to a specified range of bytes in the byte array.

```
HRESULT LockRegion(
                    //Specifies the byte offset for the beginning of the range
ULARGE_INTEGER libOffset,
                    //Specifies the length of the range in bytes
ULARGE_INTEGER cb,
    DWORD           //Specifies the type of restriction on accessing the specified range
dwLockType
);
```

Parameters

libOffset

[in]Specifies the byte offset for the beginning of the range.

cb

[in]Specifies, in bytes, the length of the range to be restricted.

dwLockType

[in]Specifies the type of restrictions being requested on accessing the range. This parameter uses one of the values from the **LOCKTYPE** enumeration.

Return Values

S_OK

The specified range of bytes was locked.

STG_E_INVALIDFUNCTION

Locking is not supported at all or the specific type of lock requested is not supported.

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

STG_E_INVALIDHANDLE

An underlying file has been prematurely closed, or the correct floppy disk has been replaced by an invalid one.

Remarks

ILockBytes::LockRegion restricts access to the specified range of bytes. Once a region is locked, attempts by others to gain access to the restricted range must fail with the STG_E_ACCESSDENIED

error.

The byte range can extend past the current end of the byte array. Locking beyond the end of an array is useful as a method of communication between different instances of the byte array object without changing data that is actually part of the byte array. For example, an implementation of **ILockBytes** for compound files could rely on locking past the current end of the array as a means of access control, using specific locked regions to indicate permissions currently granted.

The *dwLockType* parameter specifies one of three types of locking, using values from the **LOCKTYPE** enumeration. The types are as follows: locking to exclude other writers, locking to exclude other readers or writers, and locking that allows only one requestor to obtain a lock on the given range. This third type of locking is usually an alias for one of the other two lock types, and permits an Implementer to add other behavior as well. A given byte array might support either of the first two types, or both.

To determine the lock types supported by a particular **ILockBytes** implementation, you can examine the *grfLocksSupported* member of the **STATSTG** structure returned by a call to **ILockBytes::Stat**.

Any region locked with **ILockBytes::LockRegion** must later be explicitly unlocked by calling **ILockBytes::UnlockRegion** with exactly the same values for the *libOffset*, *cb*, and *dwLockType* parameters. The region must be unlocked before the stream is released. Two adjacent regions cannot be locked separately and then unlocked with a single unlock call.

Notes to Callers

Since the type of locking supported is optional and can vary in different implementations of **ILockBytes**, you must provide code to deal with the STG_E_INVALIDFUNCTION error.

Notes to Implementers

Support for this method depends on how the storage object built on top of the **ILockBytes** implementation is used. If you know that only one storage object at any given time can be opened on the storage device that underlies the byte array, then your **ILockBytes** implementation does not need to support locking. However, if multiple simultaneous openings of a storage object are possible, then region locking is needed to coordinate them.

A **LockRegion** implementation can choose to support all, some, or none of the lock types. For unsupported lock types, the implementation should return STG_E_INVALIDFUNCTION.

See Also

ILockBytes::Stat, **ILockBytes::UnlockRegion**, **IStream::LockRegion**, **LOCKTYPE**, **ILockBytes—File-Based Implementation**, **ILockBytes—Global Memory Implementation**

ILockBytes::ReadAt Quick Info

Reads a specified number of bytes starting at a specified offset from the beginning of the byte array object.

```
HRESULT ReadAt(
  ULARGE_INTEGER ulOffset,      //Specifies the starting point for reading data
  void *pv,                     //Points to the buffer into which the data is read
  ULONG cb,                     //Specifies the number of bytes to read
  ULONG *pcbRead                //Pointer to location that contains actual number of bytes read
);
```

Parameters

ulOffset

[in]Specifies the starting point from the beginning of the byte array for reading data.

pv

[in]Points to the buffer into which the byte array is read.

cb

[in]Specifies the number of bytes of data to attempt to read from the byte array.

pcbRead

[out]Pointer to a location where this method writes the actual number of bytes read from the byte array. You can set this pointer to NULL to indicate that you are not interested in this value. In this case, this method does not provide the actual number of bytes read.

Return Values

S_OK

Indicates that the specified number of bytes were read, or the maximum number of bytes were read up to the end of the byte array.

E_FAIL

Data could not be read from the byte array.

E_PENDING

Asynchronous Storage only: Part or all of the data to be read is currently unavailable. For more information see **IFillLockBytes** and [insert jump to asynchronous storage overview, which is to come].

STG_E_ACCESSDENIED

The caller does not have permission to access the byte array.

STG_E_READFAULT

The number of bytes to be read does not equal the number of bytes that were acutally read.

Remarks

ILockBytes::ReadAt reads bytes from the byte array object. It reports the number of bytes that were

actually read. This value may be less than the number of bytes requested if an error occurs or if the end of the byte array is reached during the read.

It is not an error to read less than the specified number of bytes if the operation encounters the end of the byte array. Note that this is the same end-of-file behavior as found in MS-DOS FAT file system files.

See Also

ILockBytes::WriteAt, **ILockBytes—File-Based Implementation**, **ILockBytes—Global Memory Implementation**

ILockBytes::SetSize Quick Info

Changes the size of the byte array.

```
HRESULT SetSize(
        //Specifies the new size of the byte array in bytes
ULARGE_INTEGER cb
);
```

Parameter

cb

[in]Specifies the new size of the byte array as a number of bytes.

Return Values

S_OK

The size of the byte array was successfully changed.

STG_E_ACCESSDENIED

The caller does not have permission to access the byte array.

STG_E_MEDIUMFULL

The byte array size is not changed because there is no space left on the storage device.

Remarks

ILockBytes::SetSize changes the size of the byte array. If the *cb* parameter is larger than the current byte array, the byte array is extended to the indicated size by filling the intervening space with bytes of undefined value, as does **ILockBytes::WriteAt**, if the seek pointer is past the current end-of-stream.

If the *cb* parameter is smaller than the current byte array, the byte array is truncated to the indicated size.

Notes to Callers

Callers cannot rely on STG_E_MEDIUMFULL being returned at the appropriate time because of cache buffering in the operating system or network. However, callers must be able to deal with this return code because some **ILockBytes** implementations might support it.

See Also

ILockBytes::ReadAt, **ILockBytes::WriteAt**, **ILockBytes－File-Based Implementation**, **ILockBytes－Global Memory Implementation**

ILockBytes::Stat Quick Info

Retrieves a **STATSTG** structure containing information for this byte array object.

```
HRESULT Stat(
  STATSTG *pstatstg,          //Location for STATSTG structure
  DWORD grfStatFlag           //Values taken from the STATFLAG enumeration
);
```

Parameters

pstatstg

[out]Points to a **STATSTG** structure in which this method places information about this byte array object. The pointer is NULL if an error occurs.

grfStatFlag

[in]Specifies whether this method should supply the *pwcsName* member of the **STATSTG** structure through values taken from the **STATFLAG** enumeration. If the STATFLAG_NONAME is specified, the *pwcsName* member of **STATSTG** is not supplied, thus saving a memory allocation operation. The other possible value, STATFLAG_DEFAULT, indicates that all STATSTG members be supplied.

Return Values

S_OK

The **STATSTG** structure was successfully returned at the specified location.

E_OUTOFMEMORY

The **STATSTG** structure was not returned due to a lack of memory for the name field in the structure.

STG_E_ACCESSDENIED

The **STATSTG** structure was not returned because the caller did not have access to the byte array.

STG_E_INSUFFICIENTMEMORY

The **STATSTG** structure was not returned, due to a lack of memory.

STG_E_INVALIDFLAG

The value for the *grfStateFlag* parameter is not valid.

STG_E_INVALIDPOINTER

The value for the *pStatStg* parameter is not valid.

Remarks

ILockBytes::Stat should supply information about the byte array object in a STATSTG structure.

See Also

STATFLAG, **STATSTG**, **ILockBytes–File-Based Implementation**, **ILockBytes–Global Memory Implementation**

ILockBytes::UnlockRegion Quick Info

Removes the access restriction on a previously locked range of bytes.

```
HRESULT UnlockRegion(
                    //Specifies the byte offset for the beginning of the range
ULARGE_INTEGER libOffset,
                    //Specifies the length of the range in bytes
ULARGE_INTEGER cb,
    DWORD           //Specifies the access restriction previously placed on the
dwLockType          range
    );
```

Parameters

libOffset

[in]Specifies the byte offset for the beginning of the range.

cb

[in]Specifies, in bytes, the length of the range that is restricted.

dwLockType

[in]Specifies the type of access restrictions previously placed on the range. This parameter uses a value from the **LOCKTYPE** enumeration.

Return Values

S_OK

The byte range was unlocked.

STG_E_INVALIDFUNCTION

Locking is not supported at all or the specific type of lock requested is not supported.

STG_E_LOCKVIOLATION

The requested unlock cannot be granted.

Remarks

ILockBytes::UnlockRegion unlocks a region previously locked with a call to **ILockBytes::LockRegion**. Each region locked must be explicitly unlocked, using the same values for the *libOffset*, *cb*, and *dwLockType* parameters as in the matching calls to **ILockBytes::LockRegion**. Two adjacent regions cannot be locked separately and then unlocked with a single unlock call.

See Also

ILockBytes::LockRegion, **LOCKTYPE**, **ILockBytes–File-Based Implementation**, **ILockBytes–Global Memory Implementation**

ILockBytes::WriteAt Quick Info

Writes the specified number of bytes starting at a specified offset from the beginning of the byte array.

```
HRESULT WriteAt(
  ULARGE_INTEGER ulOffset,        //Specifies the starting point for writing data
  void const *pv,                 //Points to the buffer containing the data to be written
  ULONG cb,                       //Specifies the number of bytes to write
  ULONG *pcbWritten               //Pointer to location that contains actual number of bytes written
);
```

Parameters

ulOffset

[in]Specifies the starting point from the beginning of the byte array for the data to be written.

pv

[in]Points to the buffer containing the data to be written.

cb

[in]Specifies the number of bytes of data to attempt to write into the byte array.

pcbRead

[out]Pointer to a location where this method specifies the actual number of bytes written to the byte array. You can set this pointer to NULL to indicate that you are not interested in this value. In this case, this method does not provide the actual number of bytes written.

Return Values

S_OK

Indicates that the specified number of bytes were written.

E_FAIL

A general failure occurred during the write.

E_PENDING

Asynchronous Storage only: Part or all of the data to be written is currently unavailable. For more information see **IFillLockBytes** and [insert jump to asynchronous storage overview, which is to come].

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for writing this byte array.

STG_E_WRITEFAULT

The number of bytes to be written does not equal the number of bytes that were acutally written.

STG_E_MEDIUMFULL

The write operation was not completed because there is no space left on the storage device. The actual number of bytes written is still returned in *pcbWritten*.

Remarks

ILockBytes::WriteAt writes the specified data at the specified location in the byte array. The number of bytes actually written must always be returned in *pcbWritten*, even if an error is returned. If the byte count is zero bytes, the write operation has no effect.

If *ulOffset* is past the end of the byte array and *cb* is greater than zero, **ILockBytes::WriteAt** increases the size of the byte array. The fill bytes written to the byte array are not initialized to any particular value.

See Also

ILockBytes::ReadAt, **ILockBytes::SetSize**, **ILockBytes－File-Based Implementation**, **ILockBytes－Global Memory Implementation**

ILockBytes - File-Based Implementation

Implemented on a byte array object underlying an OLE compound file storage object, and designed to read and write directly to a disk file.

When to Use

Methods of **ILockBytes** are called from the compound file implementations of **IStorage** and **IStream** on the compound file storage object created through a call to **StgCreateDocfile**, so you do not need to call them directly.

Remarks

ILockBytes::ReadAt

This method queries the wrapped pointer for the requested interface.

ILockBytes::WriteAt

This method queries the wrapped pointer for the requested interface.

ILockBytes::Flush

This method queries the wrapped pointer for the requested interface.

ILockBytes::SetSize

This method queries the wrapped pointer for the requested interface.

ILockBytes::LockRegion

The *dwLockTypes* parameter is set to LOCK_ONLYONCE OR LOCK_EXCLUSIVE, which will allow or restrict access to locked regions.

ILockBytes::UnlockRegion

This method unlocks the region locked by **ILockBytes::LockRegion**.

ILockBytes::Stat

The OLE-provided **IStorage::Stat** implementation calls the **ILockBytes::Stat** method to retrieve information about the byte array object. If there is no reasonable name for the byte array, the OLE-provided **ILockBytes::Stat** method returns NULL in the *pwcsName* field of the **STATSTG** structure.

See Also

ILockBytes, **IStorage**, **IStream**

ILockBytes - Global Memory Implementation

Implemented on a byte array object underlying an OLE compound file storage object, and designed to read and write directly to global memory.

When to Use

Methods of **ILockBytes** are called from the compound file implementations of IStorage and IStream on the compound file storage object created through a call to **StgCreateDocfile**.

Remarks

ILockBytes::ReadAt

This method queries the wrapped pointer for the requested interface.

ILockBytes::WriteAt

This method queries the wrapped pointer for the requested interface.

ILockBytes::Flush

This method queries the wrapped pointer for the requested interface.

ILockBytes::SetSize

This method queries the wrapped pointer for the requested interface.

ILockBytes::LockRegion

This implementation does not support locking, so *dwLocksType* is set to zero. It is the caller's responsibility to ensure accesses are valid and mutually exclusive.

ILockBytes::UnlockRegion

This implementation does not support locking.

ILockBytes::Stat

The OLE-provided **IStorage::Stat** implementation calls the **ILockBytes::Stat** method to retrieve information about the byte array object. If there is no reasonable name for the byte array, the OLE-provided **ILockBytes::Stat** method returns NULL in the *pwcsName* field of the **STATSTG** structure.

See Also

ILockBytes, IStorage, IStream

IMalloc Quick Info

Allocates, frees, and manages memory.

When to Implement

In general, you should not implement **IMalloc**, instead using the OLE implementation, which is guaranteed to be thread-safe in managing task memory. You get a pointer to the OLE task allocator object's **IMalloc** through a call to the **CoGetMalloc** function.

When to Use

Call the methods of **IMalloc** to allocate and manage memory. The OLE libraries and object handlers also call the **IMalloc** methods to manage memory. Object handlers should call **CoGetMalloc** to get a pointer to the **IMalloc** implementation on the task allocator object, and use the implementation of those methods to manage task memory.

The **IMalloc** methods **Alloc**, **Free**, and **Realloc** are similar to the C library functions **malloc**, **free**, and **realloc**. For debugging, refer to the functions **CoRegisterMallocSpy** and **CoRevokeMallocSpy**.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IMalloc Methods	Description
Alloc	Allocates a block of memory.
Realloc	Changes the size of a previously allocated block of memory.
Free	Frees a previously allocated block of memory.
GetSize	Returns the size in bytes of a previously allocated block of memory.
DidAlloc	Determines if this instance of IMalloc was used to allocate the specified block of memory.
HeapMinimize	Minimizes the heap by releasing unused memory to the operating system.

See Also

CoGetMalloc, IMallocSpy, CoRegisterMallocSpy, CoRevokeMallocSpy

IMalloc::Alloc Quick Info

Allocates a block of memory.

```
void * Alloc(
  ULONG cb      //Size of the requested memory block in bytes
);
```

Parameter

cb

[in] Size , in bytes, of the memory block to be allocated.

Return Values

If successful, **Alloc** returns a pointer to the allocated memory block.

NULL

If insufficient memory is available, **Alloc** returns NULL.

Remarks

The **IMalloc::Alloc** method allocates a memory block in essentially the same way that the C Library **malloc** function does.

The initial contents of the returned memory block are undefined - there is no guarantee that the block has been initialized, so you should initialize it in your code. The allocated block may be larger than *cb* bytes because of the space required for alignment and for maintenance information.

If *cb* is zero, **IMalloc::Alloc** allocates a zero-length item and returns a valid pointer to that item. If there is insufficient memory available, **IMalloc::Alloc** returns NULL.

Note Applications should always check the return value from this method, even when requesting small amounts of memory, because there is no guarantee the memory will be allocated.

See Also

IMalloc::Free, IMalloc::Realloc, CoTaskMemAlloc

IMalloc::DidAlloc Quick Info

Determines if this allocator was used to allocate the specified block of memory.

```
int DidAlloc(

  void *pv      //Pointer to the memory block
);
```

Parameter

pv

[in] Pointer to the memory block; can be a NULL pointer, in which case, -1 is returned.

Return Values

1

The memory block was allocated by this **IMalloc** instance.

0

The memory block was *not* allocated by this **IMalloc** instance.

-1

DidAlloc is unable to determine whether or not it allocated the memory block.

Remarks

Calling **IMalloc::DidAlloc** is useful if a application is using multiple allocations, and needs to know whether a previously allocated block of memory was allocated by a particular allocation.

See Also

IMalloc::Alloc, **IMalloc::HeapMinimize**, **IMalloc::Realloc**

IMalloc::Free Quick Info

Frees a previously allocated block of memory.

```
void Free(
  void * pv      //Pointer to the memory block to be freed
);
```

Parameter

pv

[in] Pointer to the memory block to be freed.

Remarks

IMalloc:Free frees a block of memory previously allocated through a call to **IMalloc::Alloc** or **IMalloc::Realloc**. The number of bytes freed equals the number of bytes that were allocated. After the call, the memory block pointed to by *pv* is invalid and can no longer be used.

Note The *pv* parameter can be NULL. If so, this method has no effect.

See Also

IMalloc::Alloc, **IMalloc::Realloc**, **CoTaskMemFree**

IMalloc::GetSize Quick Info

Returns the size (in bytes) of a memory block previously allocated with **IMalloc::Alloc** or **IMalloc::Realloc**.

```
ULONG GetSize(
  void *pv      //Pointer to the memory block for which the size is requested
);
```

Parameter

pv

[in] Pointer to the memory block for which the size is requested.

Return Value

The size of the allocated memory block in bytes or, if *pv* is a NULL pointer, -1.

Remarks

To get the size in bytes of a memory block, the block must have been previously allocated with **IMalloc::Alloc** or **IMalloc::Realloc**. The size returned is the actual size of the allocation, which may be greater than the size requested when the allocation was made.

See Also

IMalloc::Alloc, **IMalloc::Realloc**

IMalloc::HeapMinimize Quick Info

Minimizes the heap as much as possible by releasing unused memory to the operating system, coalescing adjacent free blocks and committing free pages.

void HeapMinimize();

Remarks

Calling **IMalloc::HeapMinimize** is useful when an application has been running for some time and the heap may be fragmented.

See Also

IMalloc::Alloc, **IMalloc::Free**, **IMalloc::Realloc**

IMalloc::Realloc Quick Info

Changes the size of a previously allocated memory block.

```
void *Realloc(

  void *pv,       //Pointer to memory block to be reallocated
  ULONG cb        //Size of the memory block in bytes
);
```

Parameters

pv

[in] Pointer to the memory block to be reallocated. The pointer can have a NULL value, as discussed in the following Remarks section.

cb

[in] Size of the memory block (in bytes) to be reallocated. It can be zero, as discussed in the following remarks.

Return Values

Reallocated memory block

Memory block successfully reallocated.

NULL

Insufficient memory or *cb* is zero and *pv* is not NULL.

Remarks

IMalloc::Realloc reallocates a block of memory, but does guarantee that the contents of the returned memory block are initialized. Therefore, the caller is responsible for intializing it in code, subsequent to the reallocation. The allocated block may be larger than *cb* bytes because of the space required for alignment and for maintenance information.

The *pv* argument points to the beginning of the memory block. If *pv* is NULL, **IMalloc::Realloc** allocates a new memory block in the same way that **IMalloc::Alloc** does. If *pv* is not NULL, it should be a pointer returned by a prior call to **IMalloc::Alloc**.

The *cb* argument specifies the size (in bytes) of the new block. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block can be in a different location. Because the new block can be in a different memory location, the pointer returned by **IMalloc::Realloc** is not guaranteed to be the pointer passed through the *pv* argument. If *pv* is not NULL and *cb* is zero, then the memory pointed to by *pv* is freed.

IMalloc::Realloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is NULL if the size is zero and the buffer argument is not NULL, or if there is not enough memory available to expand the block to the given size. In the first case, the original block is freed; in the second, the original block is unchanged.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

See Also

IMalloc::Alloc, IMalloc::Free

IMallocSpy Quick Info

The **IMallocSpy** interface is a debugging interface that allows application developers to monitor (spy on) memory allocation, detect memory leaks and simulate memory failure in calls to **IMalloc** methods.

Caution The **IMallocSpy** interface is intended to be used **only** to debug application code under development. Do not ship this interface to retail customers of your application, because it causes severe performance degradation and could conflict with user-installed software to produce unpredictable results.

When to Implement

Implement this interface to debug memory allocation during application development.

When to Use

When an implementation of **IMallocSpy** is registered with **CoRegisterMallocSpy**, OLE calls the pair of **IMallocSpy** methods around the corresponding **IMalloc** method. You would not make direct calls to **IMallocSpy** methods. The OLE SDK contains a sample implementation of **IMallocSpy**. The call to the pre-method through the return from the corresponding post-method is guaranteed to be thread-safe in multi-threaded operations.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IMallocSpy Methods	Description
PreAlloc	Called before invoking IMalloc::Alloc, and may extend or modify the allocation to store debug information.
PostAlloc	Called after invoking **IMalloc::Alloc**.
PreFree	Called before invoking IMalloc::Free.
PostFree	Called after invoking **IMalloc::Free**.
PreRealloc	Called before invoking IMalloc::Realloc.
PostRealloc	Called after invoking **IMalloc::Realloc**.
PreGetSize	Called before invoking IMalloc::GetSize.
PostGetSize	Called after invoking **IMalloc::GetSize**.
PreDidAlloc	Called before invoking IMalloc::DidAlloc.
PostDidAlloc	Called after invoking **IMalloc::DidAlloc**.
PreHeapMinimize	Called before invoking IMalloc::DidAlloc.
PostHeapMinimize	Called after invoking IMalloc::HeapMinimize.

See Also

IMalloc, CoGetMalloc, CoRegisterMallocSpy

IMallocSpy::PreAlloc Quick Info

Called just prior to invoking **IMalloc::Alloc**.

```
ULONG PreAlloc(

  ULONG cbRequest      //Byte count passed to IMalloc::Alloc
);
```

Parameter

cbRequest

[in] Number of bytes specified in the allocation request the caller is passing to **IMalloc::Alloc**.

Return Value

The byte count actually passed to **IMalloc::Alloc**, which should be greater than or equal to the value of *cbRequest*.

Remarks

The **IMallocSpy::PreAlloc** implementation may extend and/or modify the allocation to store debug-specific information with the allocation.

PreAlloc can force memory allocation failure by returning 0, allowing testing to ensure that the application handles allocation failure gracefully in all cases. In this case, **PostAlloc** is not called and **Alloc** returns NULL. Forcing allocation failure is effective only if *cbRequest* is not equal to 0. If **PreAlloc** is forcing failure by returning NULL, **PostAlloc** is not called. However, if **IMalloc::Alloc** encounters a real memory failure and returns NULL, **PostAlloc** is called.

The call to **PreAlloc** through the return from **PostAlloc** is guaranteed to be thread safe.

See Also

IMalloc::Alloc, **IMallocSpy::PostAlloc**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PostAlloc Quick Info

Called just after invoking **IMalloc::Alloc**, taking as input a pointer to the **IMalloc::Alloc** caller's allocation, and returning a pointer to the actual allocation.

```
void * PostAlloc(
  void * pActual      //Pointer to the allocation actually done by IMalloc::Alloc
);
```

Parameter

pActual

[in] Pointer to the allocation done by **IMalloc::Alloc**.

Return Value

A pointer to the beginning of the memory block actually allocated. This pointer is also returned to the caller of **IMalloc::Alloc**. If debug information is written at the front of the caller's allocation, this should be a forward offset from *pActual*. The value is the same as *pActual* if debug information is appended or if no debug information is attached.

Remarks

When a spy object implementing **IMallocSpy** is registered with **CoRegisterMallocSpy**, OLE calls **IMallocSpy::PostAlloc** after any call to **IMalloc::Alloc**. It takes as input a pointer to the allocation done by the call to **IMalloc::Alloc**, and returns a pointer to the beginning of the total allocation, which could include a forward offset from the other value if **IMallocSpy::Prealloc** was implemented to attach debug information to the allocation in this way. If not, the same pointer is returned, and also becomes the return value to the caller of **IMalloc::Alloc**.

See Also

IMalloc::Alloc, **IMallocSpy::PreAlloc**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PreDidAlloc Quick Info

Called by OLE just prior to invoking **IMalloc::DidAlloc**.

```
void * PreDidAlloc(
    void * pRequest,      //Pointer the caller is passing to IMalloc::DidAlloc
    BOOL fSpyed           //Whether pRequest was allocated while this spy was active
);
```

Parameters

pRequest

[in] Pointer the caller is passing to **IMalloc::DidAlloc**.

fSpyed

[in] TRUE if the allocation was done while this spy was active.

Return Value

The pointer for which allocation status is determined. This pointer is passed to **PostDidAlloc** as the *fActual* parameter.

Remarks

When a spy object implementing **IMallocSpy** is registered with **CoRegisterMallocSpy**, OLE calls this method immediately before any call to **IMalloc::DidAlloc**. This method is included for completeness and consistency – it is not anticipated that developers will implement significant functionality in this method.

See Also

IMalloc::DidAlloc, **IMallocSpy::PostDidAlloc**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PostDidAlloc Quick Info

Called just after invoking **IMalloc::DidAlloc**.

```
int PostDidAlloc(
  void * pRequest,      //Original pointer passed to IMalloc::DidAlloc
  BOOL fSpyed,          //Whether the allocation was done while this spy was active
  int fActual           //Whether pRequest was actual value used in IMalloc call
);
```

Parameters

pRequest

[in] Pointer specified in the original call to **IMalloc::DidAlloc**.

fSpyed

[in] TRUE if the allocation was done while this spy was active.

fActual

[in] Actual value returned by **IMalloc::DidAlloc**.

Return Value

The value returned to the caller of **IMalloc::DidAlloc**.

Remarks

When a spy object implementing **IMallocSpy** is registered with **CoRegisterMallocSpy**, OLE calls this method immediately after any call to **IMalloc::DidAlloc**. This method is included for completeness and consistency – it is not anticipated that developers will implement significant functionality in this method.

For convenience, *pRequest*, the original pointer passed in the call to **IMalloc::DidAlloc**, is passed to **PostDidAlloc**. In addition, the parameter *fActual* is a boolean that indicates whether this value was actually passed to **IMalloc::DidAlloc**. If not, it would indicate that **IMallocSpy::PreDidAlloc** was implemented to alter this pointer for some debugging purpose.

The *fSpyed* parameter is a boolean that indicates whether the allocation was done while the current spy object was active.

See Also

IMalloc::DidAlloc, **IMallocSpy::PreDidAlloc**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PreFree Quick Info

Called just before invoking **IMalloc::Free** to ensure that the pointer passed to **IMalloc::Free** points to the beginning of the actual allocation.

```
void * PreFree(
  void * pRequest,     //Pointer is passing to IMalloc::Free
  BOOL fSpyed          //TRUE if this memory was allocated while the spy was active
);
```

Parameters

pRequest

[in] Pointer to the block of memory that the caller is passing to **IMalloc::Free**.

fSpyed

[in] TRUE if the *pRequest* parameter of **IMallocSpy::PreFree** was allocated while the spy was installed. This value is also passed to **IMallocSpy::PostFree**.

Return Value

The actual pointer to pass to **IMalloc::Free**.

Remarks

If **IMallocSpy::PreAlloc** modified the original allocation request passed to **IMalloc::Alloc** (or **IMalloc::Realloc**), **IMallocSpy::PreFree** must supply a pointer to the actual allocation, which OLE will pass to **IMalloc::Free**. For example, if the **PreAlloc**/**PostAlloc** pair attached a header used to store debug information to the beginning of the caller's allocation, **PreFree** must return a pointer to the beginning of this header, so all of the block that was allocated can be freed.

See Also

IMalloc::Free, **IMallocSpy::PostFree**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PostFree Quick Info

Called just after invoking **IMalloc::Free**.

```
void PostFree(
  BOOL fSpyed      //Whether the memory block to be freed was allocated while the spy is active
);
```

Parameter

fSpyed

[in] TRUE if the memory block to be freed was allocated while the current spy was active, otherwise FALSE.

Remarks

When a spy object implementing **IMallocSpy** is registered with **CoRegisterMallocSpy**, OLE calls this method immediately after any call to **IMalloc::Free**. This method is included for completeness and consistency – it is not anticipated that developers will implement significant functionality in this method. On return, the *fSpyed* parameter simply indicates whether the memory was freed while the current spy was active.

See Also

IMalloc::Free, **IMallocSpy::PreFree**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PreGetSize Quick Info

Called by OLE just prior to any call to **IMalloc::GetSize**.

```
void * PreGetSize(
    void * pRequest,      //Pointer the caller is passing to IMalloc::GetSize
    BOOL fSpyed           //TRUE if allocation was done while this spy was active
);
```

Parameters

pRequest

[in] Pointer the caller is passing to **IMalloc::GetSize**.

fSpyed

[in] TRUE if the allocation was done while the spy was active.

Return Value

Pointer to the actual allocation for which the size is to be determined.

Remarks

The **PreGetSize** method receives as its *pRequest* parameter the pointer the caller is passing to **IMalloc::GetSize**. It must then return a pointer to the actual allocation, which may have altered *pRequest* in the implementation of either the **PreAlloc** or **PreRealloc** methods of **IMallocSpy**. The pointer to the true allocation is then passed to **IMalloc::GetSize** as its *pv* parameter.

IMalloc::GetSize then returns the size determined, and OLE passes this value to **IMallocSpy::PostGetSize** in *cbActual*.

Note The size determined by **IMalloc::GetSize** is the value returned by the Win32 function **HeapSize**. On Windows NT, this is the size originally requested. On Windows 95, memory allocations are done on eight-byte boundaries. For example, a memory allocation request of 27 bytes on Windows NT would return an allocation of 32 bytes and **GetSize** would return 27. On Windows 95, the same request would return an allocation of 28 bytes and **GetSize** would return 28. Implementers of **IMallocSpy::PostGetSize** cannot assume, for example, that if *cbActual* is sizeof(debug_header), that the value is the actual size of the user's allocation.

See Also

IMalloc::GetSize, **IMallocSpy::PostGetSize**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PostGetSize Quick Info

Called just after invoking **IMalloc::GetSize**.

```
ULONG PostGetSize(
    ULONG cbActual,     //Actual size of the allocation
    BOOL fSpyed         //Whether the allocation was done while a spy was active
);
```

Parameters

cbActual

[in] Actual number of bytes in the allocation, as returned by **IMalloc::GetSize**.

fSpyed

[in] TRUE if the allocation was done while a spy was active.

Return Values

The same value returned by **IMalloc::GetSize**, which is the size of the allocated memory block in bytes.

Remarks

The size determined by **IMalloc::GetSize** is the value returned by the Win32 function **HeapSize**. On Windows NT, this is the size originally requested. On Windows 95, memory allocations are done on eight-byte boundaries. For example, a memory allocation request of 27 bytes on Windows NT would return an allocation of 32 bytes and **GetSize** would return 27. On Windows 95, the same request would return an allocation of 28 bytes and **GetSize** would return 28. Implementers of **IMallocSpy::PostGetSize** cannot assume, for example, that if *cbActual* is sizeof(debug_header), that the value is the actual size of the user's allocation.

See Also

IMalloc::GetSize, **IMallocSpy::PreGetSize**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PreHeapMinimize

Quick Info

Called just prior to invoking **IMalloc::HeapMinimize**.

void PreHeapMinimize(*void***);**

Remarks

This method is included for completeness; it is not anticipated that developers will implement significant functionality in this method.

See Also

IMalloc::HeapMinimize, **IMallocSpy::PostHeapMinimize**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PostHeapMinimize Quick Info

Called just after invoking **IMalloc::HeapMinimize**.

void PostHeapMinimize(*void***);**

Remarks

When a spy object implementing **IMallocSpy** is registered with **CoRegisterMallocSpy**, OLE calls this method immediately after any call to **IMalloc::Free**. This method is included for completeness and consistency – it is not anticipated that developers will implement significant functionality in this method.

See Also

IMalloc::HeapMinimize, **IMallocSpy::PreHeapMinimize**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PreRealloc Quick Info

Called just before invoking **IMalloc::Alloc**.

```
ULONG PreRealloc(
  void * pRequest,          //Pointer the caller is passing to IMalloc::Realloc
  ULONG cbRequest,          //Byte count the caller is passing to IMalloc::Realloc
  void ** ppNewRequest,     //Indirect pointer to be reallocated
  BOOL fSpyed               //Whether the original allocation was "spyed"
);
```

Parameters

pRequest

[in] Pointer specified in the original call to **IMalloc::Realloc**, indicating the the memory block to be reallocated.

cbRequest

[in] Memory block's byte count as specified in the original call to **IMalloc::Realloc**.

ppNewRequest

[out] Indirect pointer to the actual memory block to be reallocated. This may be different from the pointer in *pRequest* if the implementation of **IMallocSpy::PreRealloc** extends or modifies the reallocation. This is an out pointer and should always be stored by **PreRealloc**.

fSpyed

[in] TRUE if the original allocation was done while the spy was active.

Return Value

The actual byte count to be passed to **IMalloc::Realloc**.

Remarks

The **IMallocSpy::PreRealloc** implementation may extend and/or modify the allocation to store debug-specific information with the allocation. Thus, the *ppNewRequest* parameter may differ from *pRequest*, a pointer to the request specified in the original call to **IMalloc::Realloc**.

PreRealloc can force memory allocation failure by returning 0, allowing testing to ensure that the application handles allocation failure gracefully in all cases. In this case, **PostRealloc** is not called and **Realloc** returns NULL. However, if **IMalloc::Realloc** encounters a real memory failure and returns NULL, **PostRealloc** is called. Forcing allocation failure is effective only if *cbRequest* is not equal to 0.

See Also

IMalloc::Realloc, **IMallocSpy::PostRealloc**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMallocSpy::PostRealloc Quick Info

Called after invoking **IMalloc::Realloc**.

```
void * PostRealloc(

  void * pActual,      //Pointer returned by IMalloc::Realloc
  BOOL fSpyed          //Whether the original allocation was "spyed"
);
```

Parameters

pActual

[in] Pointer to the memory block reallocated by **IMalloc::Realloc**.

fSpyed

[in] If TRUE, the original memory allocation was done while the spy was active.

Return Values

A pointer to the beginning of the memory block actually allocated. This pointer is also returned to the caller of **IMalloc::Realloc**. If debug information is written at the front of the caller's allocation, it should be a forward offset from *pActual*. The value should be the same as *pActual* if debug information is appended or if no debug information is attached.

See Also

IMalloc::Realloc, **IMallocSpy::PreRealloc**, **CoRegisterMallocSpy**, **CoRevokeMallocSpy**

IMarshal Quick Info

The **IMarshal** interface enables an COM object to define and manage the marshaling of its interface pointers. The alternative is to use COM's default implementation, the preferred choice in all but a few special cases (see "When to Implement").

"Marshaling" is the process of packaging data into packets for transmission to a different process or machine. "Unmarshaling" is the process of recovering that data at the receiving end. In any given call, method arguments are marshaled and unmarshaled in one direction, while return values are marshaled and unmarshaled in the other.

Although marshaling applies to all data types, interface pointers require special handling. The fundamental problem is how client code running in one address space can correctly dereference a pointer to an interface on an object residing in a different address space. COM's solution is for a client application to communicate with the original object through a surrogate object, or *proxy*, which lives in the client's process. The proxy holds a reference to an interface on the original object and hands the client a pointer to an interface on itself. When the client calls an interface method on the original object, its call is actually going to the proxy. Therefore, from the client's point of view, all calls are in-process.

On receiving a call, the proxy marshals the method arguments and, through some means of interprocess communication, such as RPC, passes them along to code in the server process, which unmarshals the arguments and passes them to the original object. This same code marshals return values for transmission back to the proxy, which unmarshals the values and passes them to the client application.

IMarshal provides methods for creating, initializing, and managing a proxy in a client process; it does not dictate how the proxy should communicate with the original object. COM's default implementation of **IMarshal** uses RPC. When you implement this interface yourself, you are free to choose any method of interprocess communication you deem to be appropriate for your application – shared memory, named pipe, window handle, RPC – in short, whatever works.

When to Implement

Implement **IMarshal** *only* when you believe that you can realize significant optimizations to COM's default implementation. In practice, this will rarely be the case. However, there are occasions where implementing **IMarshal** may be preferred:

- The objects you are writing keep their state in shared memory. In this case, both the original process and the client process uses proxies that refer to the shared memory. This type of custom marshaling is possible only if the client process is on the same machine as the original process. OLE-provided implementations of **IStorage** and **IStream** are examples of this type of custom marshaling.
- The objects you are writing are immutable, that is, their state does not change after creation. Instead of forwarding method calls to the original objects, you simply create copies of those objects in the client process. This technique avoids the cost of switching from one process to another. Some monikers are examples of immutable objects; if you are implementing your own moniker class, you should evaluate the costs and benefits of implementing **IMarshal** on your moniker objects.
- Objects that themselves are proxy objects can use custom marshaling to avoid creating proxies to proxies. Instead, the existing proxy can refer new proxies back to the original object. This capability is important for the sake of both efficiency and robustness.
- Your server application wants to manage how calls are made across the network without affecting the interface exposed to clients. For example, if an end user were making changes to a database record, the server might want to cache the changes until the user has committed them all, at which time the entire transaction would be forwarded in a single packet. Using a custom proxy would enable the caching and batching of changes in this way.

When you choose to implement **IMarshal**, you must do so for both your original object and the proxy you

create for it. When implementing the interface on either object or proxy, you simply return E_NOTIMPL for the methods that are not implemented.

COM uses your implementation of **IMarshal** in the following manner: When it's necessary to create a remote interface pointer to your object (that is, when a pointer to your object is passed as an argument in a remote function call), COM queries your object for the **IMarshal** interface. If your object implements it, COM uses your **IMarshal** implementation to create the proxy object. If your object does not implement **IMarshal**, COM uses its default implementation.

How you choose to structure the proxy is entirely up to you. You can write the proxy to use whatever mechanisms you deem appropriate for communicating with the original object. You can also create the proxy as either a stand-alone object or as part of a larger aggregation such as a handler. However you choose to structure the proxy, it must implement **IMarshal** to work at all. You must also generate a CLSID for the proxy to be returned by your implementation of **IMarshal::GetUnmarshalClass** on the original object.

When to Use

COM calls this interface as part of system-provided marshaling support. COM's calls are wrapped in calls to **CoMarshalInterface** and **CoUnmarshalInterface**. Your code typically will not need to call this interface. Special circumstances where you might choose to do so are discussed in the "Notes to Callers" section for each method.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IMarshal Methods	Description
GetUnmarshalClass	Returns CLSID of unmarshaling code.
GetMarshalSizeMax	Returns size of buffer needed during marshaling.
MarshalInterface	Marshals an interface pointer.
UnmarshalInterface	Unmarshals an interface pointer.
ReleaseMarshalData	Destroys a marshaled data packet.
DisconnectObject	Severs all connections.

See Also

IStdMarshalInfo

IMarshal::DisconnectObject Quick Info

Forcibly releases all external connections to an object. The object's server calls the object's implementation of this method prior to shutting down.

```
HRESULT DisconnectObject(
  DWORD dwReserved      //Reserved for future use
);
```

Parameter

dwReserved

[in] Reserved for future use; must be zero. To ensure compatibility with future use, **DisConnectObject** *must not* check for zero.

Return Values

The method supports the standard return value E_FAIL, as well as the following:

S_OK

The object was disconnected successfully.

Remarks

This method is implemented on the object, not the proxy.

Notes to Callers

The usual case in which this method is called occurs when an end user forcibly closes an OLE server that has one or more running objects that implement **IMarshal**. Prior to shutting down, the server calls the **CoDisconnectObject** helper function to sever external connections to *all* its running objects. For each object that implements **IMarshal**, however, this function calls **IMarshal::DisconnectObject**, so that each object that manages its own marshaling can take steps to notify its proxy that it is about to shut down.

Notes to Implementers

As part of its normal shutdown code, a server should call the **CoDisconnectObject** function, which in turn calls **IMarshal::DisconnectObject**, on each of its running objects that implements **IMarshal**.

The outcome of any implementation of this method should be to enable a proxy to respond to all subsequent calls from its client by returning RPC_E_DISCONNECTED or CO_E_OBJECTNOTCONNECTED rather than attempting to forward the calls on to the original object. It is up to the client, of course, to destroy the proxy.

If you are implementing this method for an immutable object, such as a moniker, your implementation doesn't need to do anything because such objects are typically copied whole into the client's address space. Therefore, they have neither a proxy nor a connection to the original object. For more information on marshaling immutable objects, see **IMarshal**, "When to Implement."

See Also

CoDisconnectObject

IMarshal::GetMarshalSizeMax Quick Info

Returns an upper bound on the number of bytes needed to marshal the specified interface pointer on the specified object.

```
HRESULT GetMarshalSizeMax(
  REFIID riid,                  //Reference to the identifier of the interface to be
                                //marshaled
  void *pv,                     //Interface pointer to be marshaled
  DWORD dwDestContext,          //Destination process
  void * pvDestContext,         //Reserved for future use
  DWORD mshlflags,              //Reason for marshaling
  ULONG * pSize                 //Pointer to upper-bound value
);
```

Parameters

riid

[in] Reference to the identifier of the interface to be marshaled.

pv

[in] Interface pointer to be marshaled; can be NULL.

dwDestContext

[in] Destination context where the specified interface is to be unmarshaled. Values for *dwDestContext* come from the enumeration **MSHCTX**. Currently, unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext

[in] Reserved for future use; must be NULL.

mshlflags

[in] Flag indicating whether the data to be marshaled is to be transmitted back to the client process – the normal case – or written to a global table, where it can be retrieved by multiple clients. Valid values come from the **MSHLFLAGS** enumeration.

pSize

[out] Pointer to the upper bound on the amount of data to be written to the marshaling stream.

Return Values

The method supports the standard return value E_FAIL, as well as the following:

S_OK

The maximum size was returned successfully.

E_NOINTERFACE

The specified interface was not supported.

Remarks

This method is called indirectly, in a call to **CoGetMarshalSizeMax**, by whatever code in the server process is responsible for marshaling a pointer to an interface on an object. This marshaling code is usually a stub generated by COM for one of several interfaces that can marshal a pointer to an interface implemented on an entirely different object. Examples include the **IClassFactory** and **IOleItemContainer** interfaces. For purposes of discussion, the code responsible for marshaling a pointer is here called the "marshaling stub."

To create a proxy for an object, COM requires two pieces of information from the original object: the amount of data to be written to the marshaling stream and the proxy's CLSID.

The marshaling stub obtains these two pieces of information with successive calls to **CoGetMarshalSizeMax** and **CoMarshalInterface**.

Note to Callers

The marshaling stub, through a call to **CoGetMarshalSizeMax**, calls the object's implementation of this method to preallocate the stream buffer that will be passed to **IMarshal::MarshalInterface**.

You do not explicitly call this method if you are:

- Implementing existing COM interfaces, or
- Defining your own custom interfaces, using the Microsoft Interface Definition Language (MIDL).

In both cases, the MIDL-generated stub automatically makes the call.

If you are *not* using MIDL to define your own interface (see "Writing a Custom Interface"), your marshaling stub does not have to call **GetMarshalSizeMax**, though doing so is highly recommended. An object knows better than an interface stub what the maximum size of a marshaling data packet is likely to be. Therefore, unless you are providing an automatically growing stream that is so efficient that the overhead of expanding it is insignificant, you should call this method even when implementing your own interfaces.

The value returned by this method is only guaranteed to be valid as long as the internal state of the object being marshaled does not change. Therefore, the actual marshaling should be done immediately after this function returns, or the stub runs the risk that the object, because of some change in state, might require more memory to marshal than it originally indicated.

Notes to Implementers

Your implementation of **MarshalInterface** will use this buffer to write marshaling data into the stream. If the buffer is too small, the marshaling operation will fail. Therefore, the value returned by this method *must* be a conservative estimate of the amount of data that will, in fact, be needed to marshal the interface. Violation of this requirement should be treated as a catastrophic error.

In a subsequent call to **IMarshal::MarshalInterface**, your **IMarshal** implementation cannot rely on the caller actually having called **GetMarshalSizeMax** beforehand. It must still be wary of STG_E_MEDIUMFULL errors returned by the stream and be prepared to handle them gracefully.

To ensure that your implementation of **GetMarshalSizeMax** will continue to work properly as new destination contexts are supported in the future, delegate marshaling to COM's default implementation for all *dwDestContext* values that your implementation does not understand. To delegate marshaling to COM's default implementation, call the **CoGetStandardMarshal** function.

See Also

CoGetMarshalSizeMax, **IMarshal::MarshalInterface**

IMarshal::GetUnmarshalClass Quick Info

Returns the CLSID that COM uses to locate the DLL containing the code for the corresponding proxy. COM loads this DLL to create an uninitialized instance of the proxy.

```
HRESULT GetUnmarshalClass(
  REFIID riid,                  //Reference to the identifier of the interface to be
                                //marshaled
  void * pv,                    //Interface pointer being marshaled
  DWORD dwDestContext,          //Destination process
  void * pvDestContext,         //Reserved for future use
  DWORD mshlflags,              //Reason for marshaling
  CLSID * pCid                  //Pointer to CLSID of proxy
);
```

Parameters

riid

[in] Reference to the identifier of the interface to be marshaled.

pv

[in] Pointer to the interface to be marshaled; can be NULL if the caller does not have a pointer to the desired interface.

dwDestContext

[in] "Destination context" where the specified interface is to be unmarshaled. Values for *dwDestContext* come from the enumeration **MSHCTX**. Currently, unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext

[in] Reserved for future use; must be NULL.

mshlflags

[in] Whether the data to be marshaled is to be transmitted back to the client process – the normal case – or written to a global table, where it can be retrieved by multiple clients. Valid values come from the **MSHLFLAGS** enumeration.

pCid

[out] Pointer to the CLSID to be used to create a proxy in the client process.

Return Value

Returns S_OK if successful; otherwise, S_FALSE.

Remarks

This method is called by whatever code in the server process may be responsible for marshaling a pointer to an interface on an object. This marshaling code is usually a stub generated by COM for one of several interfaces that can marshal a pointer to an interface implemented on an entirely different object. Examples include the **IClassFactory** and **IOleItemContainer** interfaces. For purposes of this discussion, the code responsible for marshaling a pointer is called the "marshaling stub."

To create a proxy for an object, COM requires two pieces of information from the original object: the amount of data to be written to the marshaling stream and the proxy's CLSID.

The marshaling stub obtains these two pieces of information with successive calls to **CoGetMarshalSizeMax** and **CoMarshalInterface**.

Note to Callers

The marshaling stub calls the object's implementation of this method to obtain the CLSID to be used in creating an instance of the proxy. The client, upon receiving the CLSID, loads the DLL listed for it in the system registry.

You do not explicitly call this method if you are:

- Implementing existing COM interfaces, or
- Defining your own interfaces using the Microsoft Interface Definition Language (MIDL).

In both cases, the stub automatically makes the call. See "Writing a Custom Interface."

If you are not using MIDL to define your own interface, your stub must call this method, either directly or indirectly, to get the CLSID that the client-side COM Library needs to create a proxy for the object implementing the interface.

If the caller has a pointer to the interface to be marshaled, it should, as a matter of efficiency, use the *pv* parameter to pass that pointer. In this way, an implementation that may use such a pointer to determine the appropriate CLSID for the proxy does not have to call **IUnknown::QueryInterface** on itself. If a caller does not have a pointer to the interface to be marshaled, it can pass NULL.

Notes to Implementers

COM calls **GetUnmarshalClass** to obtain the CLSID to be used for creating a proxy in the client process. The CLSID to be used for a proxy is normally *not* that of the original object (see "Notes to Implementers" for the exception), but one you will have generated (using the GUIDGEN.EXE tool supplied with the Win32 SDK) specifically for your proxy object.

Implement this method for each object that provides marshaling for one or more of its interfaces. The code responsible for marshaling the object writes the CLSID, along with the marshaling data, to a stream; COM extracts the CLSID and data from the stream on the receiving side.

If your proxy implementation consists simply of copying the entire original object into the client process, thereby eliminating the need to forward calls to the original object, the CLSID returned would be the same as that of the original object. This strategy, of course, is advisable only for objects that are not expected to change.

If the *pv* parameter is NULL and your implementation needs an interface pointer, it can call **IUnknown::QueryInterface** on the current object to get it. The *pv* parameter exists merely to improve efficiency.

To ensure that your implementation of **GetUnmarshalClass** continues to work properly as new destination contexts are supported in the future, delegate marshaling to COM's default implementation for all *dwDestContext* values that your implementation does not handle. To delegate marshaling to COM's default implementation, call the **CoGetStandardMarshal** function.

IMarshal::MarshalInterface Quick Info

Writes into a stream the data required to initialize a proxy object in some client process.

```
HRESULT MarshalInterface(
  IStream *pStm,          //Pointer to stream used for marshaling
  REFIID riid,            //Reference to the identifier of the interface to be
                          marshaled
  void *pv,               //Interface pointer to be marshaled
  DWORD dwDestContext,    //Destination context
  void *pvDestContext,    //Reserved for future use
  DWORD mshlflags         //Reason for marshaling
);
```

Parameters

pStm

[in] Pointer to the stream to be used during marshaling.

riid

[in] Reference to the identifier of the interface to be marshaled. This interface must be derived from the **IUnknown** interface.

pv

[in] Pointer to the interface pointer to be marshaled; can be NULL if the caller does not have a pointer to the desired interface.

dwDestContext

[in] Destination context where the specified interface is to be unmarshaled. Values for *dwDestContext* come from the enumeration **MSHCTX**. Currently, unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext

[in] Reserved for future use; must be zero.

mshlflags

[in] Whether the data to be marshaled is to be transmitted back to the client process – the normal case – or written to a global table, where it can be retrieved by multiple clients. Valid values come from the **MSHLFLAGS** enumeration.

Return Values

The method supports the standard return value E_FAIL, as well as the following:

S_OK

The interface pointer was marshaled successfully.

E_NOINTERFACE

The specified interface is not supported.

STG_E_MEDIUMFULL

The stream is full.

Remarks

This method is called indirectly, in a call to **CoMarshalInterface**, by whatever code in the server process may be responsible for marshaling a pointer to an interface on an object. This marshaling code is usually a stub generated by COM for one of several interfaces that can marshal a pointer to an interface implemented on an entirely different object. Examples include the **IClassFactory** and **IOleItemContainer** interfaces. For purposes of this discussion, the code responsible for marshaling a pointer is called the "marshaling stub."

Notes to Callers

Normally, rather than calling **IMarshal::MarshalInterface** directly, your marshaling stub instead should call the **CoMarshalInterface** function, which contains a call to this method. The stub makes this call to command an object to write its marshaling data into a stream. The stub then either passes the marshaling data back to the client process or writes it to a global table, where it can be unmarshaled by multiple clients. The stub's call to **CoMarshalInterface** is normally preceded by a call to **CoGetMarshalSizeMax**, to get the maximum size of the stream buffer into which the marshaling data will be written.

You do not explicitly call this method if you are:

- Implementing existing COM interfaces, or
- Defining your own interfaces using the Microsoft Interface Definition Language (MIDL).

In both cases, the MIDL-generated stub automatically makes the call.

If you are not using MIDL to define your own interface, your marshaling stub must call this method, either directly or indirectly.Your stub implementation should call **MarshalInterface** immediately after its previous call to **IMarshal::GetMarshalSizeMax** returns. Because the value returned by **GetMarshalSizeMax** is guaranteed to be valid only so long as the internal state of the object being marshaled does not change, a delay in calling **MarshalInterface** runs the risk that the object will require a larger stream buffer than originally indicated.

If the caller has a pointer to the interface to be marshaled, it should, as a matter of efficiency, use the *pv* parameter to pass that pointer. In this way, an implementation that may use such a pointer to determine the appropriate CLSID for the proxy does not have to call **IUnknown::QueryInterface** on itself. If a caller does not have a pointer to the interface to be marshaled, it can pass NULL.

Notes to Implementers

Your implementation of **IMarshal::MarshalInterface** must write to the stream whatever data is needed to initialize the proxy on the receiving side. Such data would include a reference to the interface to be marshaled, a **MSHLFLAGS** value specifying whether the data should be returned to the client process or written to a global table, and whatever is needed to connect to the object, such as a named pipe, handle to a window, or pointer to an RPC channel.

Your implementation should not assume that the stream is large enough to hold all the data. Rather, it should gracefully handle a STG_E_MEDIUMFULL error. Just before exiting, your implementation should position the seek pointer in the stream immediately after the last byte of data written.

If the *pv* parameter is NULL and your implementation needs an interface pointer, it can call **IUnknown::QueryInterface** on the current object to get it. The *pv* parameter exists merely to improve efficiency.

To ensure that your implementation of **MarshalInterface** continues to work properly as new destination contexts are supported in the future, delegate marshaling to COM's default implementation for all *dwDestContext* values that your implementation does not handle. To delegate marshaling to COM's default implementation, call the **CoGetStandardMarshal** helper function.

Using the **MSHLFLAGS** enumeration, callers can specify whether an interface pointer is to be marshaled back to a single client or written to a global table, where it can be unmarshaled by multiple clients. You must make sure that your object can handle calls from the multiple proxies that might be created from the same initialization data.

See Also

CoGetStandardMarshal, **IMarshal::GetMarshalSizeMax**, **IMarshal::GetUnmarshalClass**, **IMarshal::UnmarshalInterface**, **Writing a Custom Interface**

IMarshal::ReleaseMarshalData Quick Info

Destroys a marshaled data packet.

```
HRESULT ReleaseMarshalData(
  IStream * pStm      //Pointer to stream used for unmarshaling
);
```

Parameter

pStm

[in] Pointer to a stream that contains the data packet to be destroyed.

Return Values

The method supports the standard return value E_FAIL, as well as the following:

S_OK

The data packet was released successfully.

IStream errors

This function can also return any of the stream-access error values for the **IStream** interface.

Remarks

If an object's marshaled data packet does not get unmarshaled in the client process space, and the packet is no longer needed. The client calls **ReleaseMarshalData** on the proxy's **IMarshal** implementation to instruct the object to destroy the data packet. The call occurs within the **CoReleaseMarshalData** function. The data packet serves as an additional reference on the object, and releasing the data is like releasing an interface pointer by calling **IUnknown::Release**.

If the marshaled data packet somehow does not arrive in the client process, or **ReleaseMarshalData** is not successfully re-created in the proxy, COM can call this method on the object itself.

Notes to Callers

You will rarely if ever have occasion to call this method yourself. A possible exception would be if you were to implement **IMarshal** on a class factory for a class object on which you are also implementing **IMarshal**. In this case, if you are marshaling the object to a table, where it can be retrieved by multiple clients, you might, as part of your unmarshaling routine, call **ReleaseMarshalData** to release the data packet for each proxy.

Notes to Implementers

If your implementation stores state information about marshaled data packets, you can use this method to release the state information associated with the data packet represented by *pStm*. Your implementation should also position the seek pointer in the stream past the last byte of data.

See Also

CoUnMarshalInterface, **CoReleaseMarshalData**

IMarshal::UnmarshalInterface Quick Info

Initializes a newly created proxy and returns an interface pointer to that proxy.

```
HRESULT UnmarshalInterface(
  IStream * pStm,     //Pointer to the stream to be unmarshaled
  REFIID riid,        //Reference to the identifier of the interface to be
                      unmarshaled
  void ** ppv         //Indirect pointer to interface
);
```

Parameters

pStm

[in] Pointer to the stream from which the interface pointer is to be unmarshaled.

riid

[in] Reference to the identifier of the interface to be unmarshaled.

ppv

[out] Indirect pointer to the interface.

Return Values

The method supports the standard return value E_FAIL, as well as the following:

S_OK

The interface pointer was unmarshaled successfully.

E_NOINTERFACE

The specified interface was not supported.

Remarks

The COM library in the process where unmarshaling is to occur calls the proxy's implementation of this method.

Notes to Callers

You do not call this method directly. There are, however, some situations in which you might call it indirectly through a call to **CoUnmarshalInterface**. For example, if you are implementing a stub, your implementation would call **CoUnmarshalInterface** when the stub receives an interface pointer as a parameter in a method call.

Notes to Implementers

The proxy's implementation should read the data written to the stream by the original object's implementation of **IMarshal::MarshalInterface** and use that data to initialize the proxy object whose CLSID was returned by the marshaling stub's call to the original object's implementation of **IMarshal::GetUnmarshalClass**.

To return the appropriate interface pointer, the proxy implementation can simply call

IUnknown::QueryInterface on itself, passing the *riid* and *ppv* parameters. However, your implementation of **UnmarshalInterface** is free to create a different object and, if necessary, return a pointer to it.

Just before exiting, even if exiting with an error, your implementation should reposition the seek pointer in the stream immediately after the last byte of data read.

See Also

IMarshal::GetUnmarshalClass, **IMarshal::MarshalInterface**

IMarshal - Default Implementation

COM uses its own internal implementation of the **IMarshal** interface to marshal any object that does not provide its own implementation. COM makes this determination by querying the object for **IMarshal**. If the interface is missing, COM defaults to its internal implementation.

COM's default implementation of **IMarshal** uses a generic proxy for each object, and creates individual stubs and proxies, as they are needed, for each interface implemented on the object. This mechanism is necessary because COM cannot know in advance what particular interfaces a given object may implement. Developers who do not use COM's default marshaling, electing instead to write their own proxy and marshaling routines, know at compile time all the interfaces to be found on their objects and therefore understand exactly what marshaling code is required. COM, in providing marshaling support for all objects, must do so at run time.

The *interface proxy* resides in the client process; the *interface stub*, in the server. Together, each pair handles all marshaling for its interface. The job of each *interface proxy* is to marshal arguments and unmarshal return values and out parameters that are passed back and forth in subsequent calls to its interface. The job of each *interface stub* is to unmarshal function arguments and pass them along to the original object, then marshal the return values and out parameters that the object returns.

Proxy and stub communicate by means of an RPC (remote procedure call) channel, which utilizes the system's RPC infrastructure for interprocess communication. The RPC channel implements a single interface **IRpcChannelBuffer,** an internal interface to which both interface proxies and stubs hold a pointer. The proxy and stub call the interface to obtain a marshaling packet, send the data to their counterpart, and destroy the packet when they are done. The interface stub also holds a pointer to the original object.

For any given interface, the proxy and stub are both implemented as instances of the same class, which is listed for each interface in the system registry under the label *ProxyStubClsid32* (or *ProxyStubClsid* on 16-bit systems). This entry maps the interface's IID to the CLSID of its proxy and stub objects. When COM needs to marshal an interface, it looks in the system registry to obtain the appropriate CLSID. The server identified by this CLSID implements both the interface proxy and interface stub.

Most often, the class to which this CLSID refers is automatically generated by a tool whose input is a description of the function signatures and semantics of a given interface, written in some interface description language. While using such a language is highly recommended and encouraged for accuracy's sake, doing so is by no means required. Proxies and stubs are merely Component Object Model components used by the RPC infrastructure and, as such, can be written in any manner desired so long as the correct external contracts are upheld. The programmer who designs a new interface is responsible for ensuring that all interface proxies and stubs that ever exist agree on the representation of their marshaled data.

When created, interface proxies are always aggregated into a larger proxy, which represents the object as a whole. This object proxy also aggregates COM's generic proxy object, which is known as the *proxy manager*. The proxy manager implements two interfaces: **IUnknown** and **IMarshal.** All of the other interfaces that may be implemented on an object are exposed in its object proxy through the aggregation of individual interface proxies. A client holding a pointer to the object proxy "believes" it holds a pointer to the actual object.

A proxy representing the object as a whole is required in the client process so that a client can distinguish calls to the same interfaces implemented on entirely different objects. Such a requirement does not exist in the server process, however, where the object itself resides, because all interface stubs communicate only with the objects for which they were created. No other connection is possible.

Interface stubs, by contrast with interface proxies, are not aggregated, because there is no need that they appear to some external client to be part of a larger whole. When connected, an interface stub is given a pointer to the server object to which it should forward method invocations that it receives. Although it is useful to refer conceptually to a "stub manager," meaning whatever pieces of code and state in the server-side RPC infrastructure that service the remoting of a given object, there is no direct requirement that the code and state take any particular, well-specified form.

The first time a client requests a pointer to an interface on a particular object, COM loads an **IClassFactory** stub in the server process and uses it to marshal the first pointer back to the client. In the client process, COM loads the generic proxy for the class factory object and calls its implementation of **IMarshal** to unmarshal that first pointer. COM then creates the first interface proxy and hands it a pointer to the RPC channel. Finally, COM returns the **IClassFactory** pointer to the client, which uses it to call **IClassFactory::CreateInstance**, passing it a reference to the interface.

Back in the server process, COM now creates a new instance of the object, along with a stub for the requested interface. This stub marshals the interface pointer back to the client process, where another object proxy is created, this time for the object itself. Also created is a proxy for the requested interface, a pointer to which is returned to the client. With subsequent calls to other interfaces on the object, COM will load the appropriate interface stubs and proxies as needed.

When a new interface proxy is created, COM hands it a pointer to the proxy manager's implementation of **IUnknown**, to which it delegates all **QueryInterface** calls. Each interface proxy implements two interfaces of its own: the interface it represents and **IRpcProxyBuffer**. The interface proxy exposes its own interface directly to clients, which can obtain its pointer by calling **QueryInterface** on the proxy manager. Only COM, however, can call **IRpcProxyBuffer,** which it uses to connect and disconnect the proxy to the RPC channel. A client cannot query an interface proxy to obtain a pointer to the **IRpcProxyBuffer** interface.

On the server side, each interface stub implements **IRpcStubBuffer**, an internal interface. The server code acting as a stub manager calls **IRpcStubBuffer::Connect** and passes the interface stub the **IUnknown** pointer of its object.

When an interface proxy receives a method invocation, it obtains a marshaling packet from its RPC channel through a call to **IRpcChannelBuffer::GetBuffer**. The process of marshaling the arguments will copy data into the buffer. When marshaling is complete, the interface proxy invokes **IRpcChannelBuffer::SendReceive** to send the marshaled packet to the corresponding interface stub. When **IRpcChannelBuffer::SendReceive** returns, the buffer into which the arguments were marshaled will have been replaced by a new buffer containing the return values marshaled from the interface stub. The interface proxy unmarshals the return values, invokes **IRpcChannelBuffer::FreeBuffer** to free the buffer, then returns the return values to the original caller of the method.

It is the implementation of **IRpcChannelBuffer::SendReceive** that actually sends the request to the server process and that knows how to identify the server process and, within that process, the object to which the request should be sent. The channel implementation also knows how to forward the request on to the appropriate stub manager in that process. The interface stub unmarshals the arguments from the provided buffer, invokes the indicated method on the server object, and marshals the return values back into a new buffer, allocated by a call to **IRpcchannelBuffer::GetBuffer**. The channel then transmits the return data packet back to the interface proxy, which is still in the middle of **IRpcchannelBuffer::SendReceive**, which returns to the interface proxy.

A particular instance of an interface proxy can be used to service more than one interface, so long as two conditions are met. First, the IIDs of the affected interfaces must be mapped to the the appropriate *ProxyStubClsid* in the system registry. Second, the interface proxy must support calls to **QueryInterface** from one supported interface to the other interfaces, as usual, as well as from **IUnknown** and **IRpcProxyBuffer**.

A single instance of an interface stub can also service more than one interface, but only if that set of interfaces has a strict single-inheritance relationship. This restriction exists because the stub can direct method invocations to multiple interfaces only where it knows in advance which methods are implemented on which interfaces.

Both proxies and stubs will at various times have need to allocate or free memory. Interface proxies, for example, will need to allocate memory in which to return out parameters to their caller. In this respect, interface proxies and interface stubs are just normal COM components, in that they should use the standard task allocator (see **CoGetMalloc**).

When to Use

You should use COM's default implementation of **IMarshal** except in those very few cases where your application has special requirements that COM's default implementation does not address, or where you can achieve optimizations over the marshaling code COM has provided. For examples of such special cases, see **IMarshal**, "When to Implement."

See Also

IMarshal

IMessageFilter Quick Info

The **IMessageFilter** interface provides OLE servers and applications with the ability to selectively handle incoming and outgoing OLE messages while waiting for responses from synchronous calls. Filtering messages helps to ensure that calls are handled in a manner that improves performance and avoids deadlocks. OLE messages can be synchronous, asynchronous, or input-synchronized; the majority of interface calls are synchronous.

Synchronous calls require the caller to wait for a reply before continuing. OLE enters a modal loop while waiting for the reply. During this time, the caller is still able to receive and dispatch incoming messages.

Asynchronous calls allow the caller to proceed without waiting for a response from the called object. Today, in OLE, the only asynchronous calls are to an object's **IAdviseSink** interface. While the object is processing an asynchronous call, it is prohibited from making any synchronous calls back to the calling object.

Input-synchronized calls require the called object to complete the call before relinquishing control, ensuring that behaviors such as focus management and type-ahead function correctly.

When to Implement

You will probably want to implement your own message filter. The default implementation provided by COM offers only minimal message filtering capability. Although message filtering is no longer as significant as it was with 16-bit applications, since the size of the message queue is now virtually unlimited, you still should implement **IMessageFilter** as a way of resolving deadlocks.

COM will call your implementation of **IMessageFilter** to find out if an application is blocking, so that you can task-switch to that application and give the user an opportunity to deal with the situation. For example, if you have Microsoft Word talking to Microsoft Excel, with Excel running in the background in formula mode, in which formulas are being applied to data on the worksheet to compute different or "what if" results, Excel won't check all calls, thereby blocking further action. **IMessageFilter** would put up a dialog box indicating which task was blocking and provide the user with an opportunity to deal with the deadlock.

Although it is probably obvious from the method descriptions, it may still be useful to point out that **HandleIncomingCall** is an object-based method and **RetryRejectedCall** and **MessagePending** are client-based methods. Clearly, the object must have some way of handling incoming calls from external clients. **HandleIncomingCall** provides that functionality by allowing the object to handle or defer some incoming calls and reject others. The client also needs to know how an object is going to handle its call. so that it can respond appropriately. The client needs to know if a call has been rejected, or just deferred temporarily, so that it can retry rejected calls after some specified time. The client also needs to be able to respond to Windows messages, while at the same time waiting for replies to pending messages.

You will use **CoRegisterMessageFilter** to register your message filter. Once registered, COM then calls your message filter instead of the default implementation.

When to Use

You don't call this interface directly. It's provided by the OLE server or application and called by the COM.

Application Shutdown with WM_QUERYENDSESSION and WM_ENDSESSION

When a user exits Windows, each open application receives a WM_QUERYENDSESSION message followed by a WM_ENDSESSION message, provided the exit is not canceled. These messages are invoked with **SendMessage**, which unfortunately restricts the initiation of all outgoing LRPC calls. This is a problem for container applications that have open embedded objects when they receive the shutdown request because LRPC is needed to close those objects.

Container and container/server applications with open documents typically display a message box on receipt of the WM_QUERYENDSESSION message that asks if the user wants to save changes before exiting. A positive response is usually the default. The recommendation for dealing with the situation described above is for the application to display an alternate message box asking if the user wants to discard changes; a negative response should be the default. If the user chooses to discard the changes, TRUE should be returned for WM_QUERYENDSESSION, which .signals to Windows that it can terminate. If the user does not want to discard the changes, FALSE should be returned. No attempt should be made to close or release running embeddings.

Server applications should return TRUE for WM_QUERYENDSESSION without prompting the user.On receipt of a WM_ENDSESSION message, all OLE applications should execute the normal close sequence for each application's documents and/or objects. At the same time, you should ignore any errors resulting from any cross-process calls or calls to **IUnknown::Release**. All storage pointers (**IStorage** and **IStream** interface pointers) must be released to properly flush any temporary files maintained by the compound file implementation of structured storage.

See Also

CoRegisterMessageFilter

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IMessageFilter Methods	**Description**
HandleIncomingCall	Provides a single entry point for incoming calls.
RetryRejectedCall	Provides application with opportunity to display a dialog box offering retry or cancel or task switching options.
MessagePending	Indicates a Windows message has arrived while OLE is waiting to respond to a remote call.

See Also

OleUIBusy, **OLEUIBUSY**, In the *WIN32 SDK*: "Messages and Message Queues"

IMessageFilter::HandleInComingCall Quick Info

An object-based method that provides the ability to filter or reject incoming calls (or call backs) to an object or a process. This method is called prior to each method invocation originating outside the current process.

```
DWORD HandleInComingCall(
  DWORD dwCallType,          //Type of incoming call
  HTASK threadIDCaller,      //Task handle calling this task
  DWORD dwTickCount,         //Elapsed tick count
  LPINTERFACEINFO lpInterfaceInfo   //Pointer to INTERFACEINFO structure
);
```

Parameters

dwCallType

[in] Kind of incoming call that has been received. Valid values are from the enumeration **CALLTYPE**. See the chapter on enumerations for details.

threadIDCaller

[in] Handle of the task calling this task.

dwTickCount

[in] Elapsed tick count since the outgoing call was made if *dwCallType* is not CALLTYPE_TOPLEVEL. If *dwCallType* is CALLTYPE_TOPLEVEL, *dwTickCount* should be ignored.

lpInterfaceInfo

[in] Pointer to an **INTERFACEINFO** structure, which identifies the object, the interface, and the method making the call. In the case of DDE calls, *lpInterfaceInfo* can be NULL because the DDE layer does not return interface information.

Return Values

SERVERCALL_ISHANDLED

The application might be able to process the call.

SERVERCALL_REJECTED

The application cannot handle the call due to an unforeseen problem, such as network unavailability, or if it is in the process of terminating.

SERVERCALL_RETRYLATER

The application cannot handle the call at this time. For example, an application might return this value when it is in a user-controlled modal state.

Remarks

If implemented, **IMessageFilter::HandleInComingCall** is called by OLE when an incoming OLE message is received.

Depending on an application's current state, a call is either accepted and processed or rejected

(permanently or temporarily). If SERVERCALL_ISHANDLED is returned, the application may be able to process the call, though success depends on the interface for which the call is destined. If the call cannot be processed, OLE returns RPC_E_CALL_REJECTED.

Input-synchronized and asynchronous calls are dispatched even if the application returns SERVERCALL_REJECTED or SERVERCALL_RETRYLATER.

IMessageFilter::HandleInComingCall should not be used to hold off updates to objects during operations such as band printing. For that purpose, use **IViewObject::Freeze**.

You can also use **IMessageFilter::HandleInComingCall** to set up the application's state so the call can be processed in the future.

See Also

IViewObject::Freeze, **CALLTYPE**, **INTERFACEINFO**

IMessageFilter::MessagePending Quick Info

A client-based method called by COM when a Windows message appears in an OLE application's message queue while the application is waiting for a reply to a remote call. Handling input while waiting for an outgoing call to finish can introduce complications. The application should determine whether to process the message without interrupting the call, continue waiting, or cancel the operation.

```
DWORD MessagePending(
  HTASK threadIDCallee,      //Called applications task handle
  DWORD dwTickCount,         //Elapsed tick count
  DWORD dwPendingType        //Call type
);
```

Parameters

threadIDCallee

[in] Task handle of the called application that has not yet responded.

dwTickCount

[in] Number of ticks since the call was made. It is calculated from the Windows **GetTickCount** function.

dwPendingType

[in] Type of call made during which a message or event was received. Valid values are from the enumeration **PENDINGTYPE** (where PENDINGTYPE_TOPLEVEL means the outgoing call was not nested within a call from another application and PENDINTGYPE_NESTED means the outgoing call was nested within a call from another application).

Return Values

PENDINGMSG_CANCELCALL

Cancel the outgoing call. This should be returned only under extreme conditions. Canceling a call that has not replied or been rejected can create orphan transactions and lose resources. OLE fails the original call and returns RPC_E_CALL_CANCELLED.

PENDINGMSG_WAITNOPROCESS

Continue waiting for the reply and do not dispatch the message unless it is a task-switching or window-activation message. A subsequent message will trigger another call to **IMessageFilter::MessagePending**. Leaving messages or events in the queue enables them to be processed normally, if the outgoing call is completed. Note that returning PENDINGMSG_WAITNOPROCESS can cause the message queue to fill.

PENDINGMSG_WAITDEFPROCESS

Because of the increased resources available in 32-bit systems, you are unlikely to get this return value. It now indicates the same state as PENDINGMSG_WAITNOPROCESS.

Keyboard and mouse messages are no longer dispatched, as was done with PENDINGMSG_WAITDEFPROCESS. However there are some cases where mouse and keyboard messages could cause the system to deadlock, and in these cases, mouse and keyboard messages are discarded. WM_PAINT messages are dispatched. Task-switching and activation messages are handled as before.

Remarks

OLE calls **IMessageFilter::MessagePending** after an application has made an OLE method call and a Windows message occurs before the call has returned. A Windows message is sent, for example, when the user selects a menu command or double-clicks an object. Before OLE makes the **IMessageFilter::MessagePending** call, it calculates the elapsed time since the original OLE method call was made. OLE delivers the elapsed time in the *dwTickCount* parameter. In the meantime, OLE does not remove the message from the queue.

Windows messages that appear in the caller's queue should remain in the queue until sufficient time has passed to ensure that the messages are probably not the result of typing ahead, but are, instead, an attempt to get attention. Set the delay with the *dwTickCount* parameter − a two- or three-second delay is recommended. If that amount of time passes and the call has not been completed, the caller should flush the messages from the queue, and the OLE UI busy dialog box should be displayed offering the user the choice of retrying the call (continue waiting) or switching to the task identified by the *threadIDCallee* parameter. This ensures that:

- If calls are completed in a reasonable amount of time, type ahead will be treated correctly.
- If the callee does not respond, type ahead is not misinterpreted and the user is able to act to solve the problem. For example, OLE 1 servers can queue up requests without responding when they are in modal dialog boxes.

Handling input while waiting for an outgoing call to finish can introduce complications. The application should determine whether to process the message without interrupting the call, continue waiting, or cancel the operation.

When there is no response to the original OLE call, the application can cancel the call and restore the OLE object to a consistent state by calling **IStorage::Revert** on its storage. The object can be released when the container can shut down. However, canceling a call can create orphaned operations and resource leaks. Canceling should be used *only* as a last resort. It is strongly recommended that applications not allow such calls to be canceled.

See Also

IStorage::Revert, **IStorage::Revert**, **OleUIBusy**, **OLEUIBUSY**, In the *WIN32 SDK*: **GetTickCount**

IMessageFilter::RetryRejectedCall Quick Info

A client-based method that gives the application an opportunity to display a dialog box so the user can retry or cancel the call, or switch to the task identified by *threadIDCallee.*

```
DWORD RetryRejectedCall(
  HTASK threadIDCallee,     //Server task handle
  DWORD dwTickCount,        //Elapsed tick count
  DWORD dwRejectType        //Returned rejection message
);
```

Parameters

threadIDCallee

[in] Handle of the server task that rejected the call.

dwTickCount

[in] Number of elapsed ticks since the call was made.

dwRejectType

[in] Specifies either SERVERCALL_REJECTED or SERVERCALL_RETRYLATER, as returned by the object application.

Return Values

-1

The call should be canceled. OLE then returns RPC_E_CALL_REJECTED from the original method call.

Value >= 0 and <100

The call is to be retried immediately.

Value >= 100

OLE will wait for this many milliseconds and then retry the call.

Remarks

OLE calls **RetryRejectedCall** on the caller's **IMessageFilter** immediately after receiving SERVERCALL_RETRYLATER or SERVERCALL_REJECTED from the **IMessageFilter::HandleInComingCall** method on the callee's **IMessageFilter**.

If a called task rejects a call, the application is probably in a state where it cannot handle such calls, possibly only temporarily. When this occurs, OLE returns to the caller and issues **IMessageFilter::RetryRejectedCall** to determine if it should retry the rejected call.

Applications should silently retry calls that have returned with SERVERCALL_RETRYLATER. If, after a reasonable amount of time has passed, say about 30 seconds, the application should display the busy dialog box; a standard implementation of this dialog box is available in the OLEDLG library. The callee may momentarily be in a state where calls can be handled. The option to wait and retry is provided for special kinds of calling applications, such as background tasks executing macros or scripts, so that they can retry the calls in a nonintrusive way.

If, after a dialog box is displayed, the user chooses to cancel, **RetryRejectedCall** returns -1 and the call will appear to fail with RPC_E_CALL_REJECTED.

IMoniker Quick Info

The **IMoniker** interface contains methods that allow you to use a moniker object, which contains information that uniquely identifies a COM object. An object that has a pointer to the moniker object's **IMoniker** interface can locate, activate, and get access to the identified object without having any other specific information on where the object is actually located in a distributed system.

Like a path to a file in a file system, a moniker contains information that allows a COM object to be located and activated. Monikers can identify any type of COM object, from a document object stored in a file to a selection within an embedded object. OLE provides a set of moniker classes that allow you to create moniker objects identifying the objects most commonly found in the system. For example, there might be an object representing a range of cells in a spreadsheet which is itself embedded in a text document stored in a file. In a distributed system, this object's moniker would identify the location of the object's system, the file's physical location on that system, the storage of the embedded object within that file, and, finally, the location of the range of cells within the embedded object.

A moniker object supports the **IMoniker** interface, which is derived from the **IPersistStream** interface, and uniquely identifies a single object in the system. Once an object providing a moniker has created the moniker object, this information cannot be changed within that object. If the moniker provider changes the information, it can only do so by creating a new moniker object, which would then uniquely identify the object in question.

Monikers have two important capabilites:

- Monikers can be saved to a persistent storage. When a moniker is loaded back into memory, it still identifies the same object.
- Monikers support an operation called "binding," which is the process of locating the object named by the moniker, activating it (loading it into memory) if it is not already active, and returning a pointer to a requested interface on that object.

Monikers are used as the basis for linking in OLE. A linked object contains a moniker that identifies its source. When the user activates the linked object to edit it, the moniker is bound; this loads the link source into memory.

When to Implement

Implement **IMoniker** only if you are writing a new moniker class. This is necessary only if you need to identify objects that cannot be identified using one of the OLE-supplied moniker classes described below.

The OLE-supplied moniker classes are sufficient for most situations. Before considering writing your own moniker class, you should make sure that your requirements cannot be satisified by these classes.

If you decide you need to write your own implementation of **IMoniker**, you must also implement the **IROTData** interface on your moniker class. This interface allows your monikers to be registered with the Running Object Table (ROT).

When to Use

Two kinds of objects call the methods of **IMoniker**:

- A component that contains one or more objects to be identified with a moniker and must provide the moniker to other objects
- A client object that must bind to the object identified by the moniker

The component providing a moniker makes it accessible to other objects. It is important to understand the differences between the various system-supplied moniker classes to know which are appropriate for a given object. OLE also provides functions for creating monikers using the OLE-supplied moniker classes.

- **File monikers** – based on a path in the file system. File monikers can be used to identify objects that are saved as files. The associated creation function is **CreateFileMoniker**.
- **Item monikers** – based on a string that identifies an object in a container. Item monikers can be used to identify objects smaller than a file, such as embedded objects in a compound document and pseudo-objects (like a range of cells in a spreadsheet). The associated creation function is **CreateItemMoniker**.
- **Generic composite monikers** – consists of two or more monikers of arbitrary type that have been composed together. Generic composite monikers allow monikers of different classes to be used in combination. The associated creation function is **CreateGenericComposite**.
- **Anti-monikers**– the inverse of file, item, or pointer monikers. Anti-monikers are used primarily for constructing relative monikers, which are analogous to relative path (such as "..\backup\report.old"), and which specify a location of an object *relative* to the location of another object). The associated creation function is **CreateAntiMoniker**.
- **Pointer monikers** – a non-persistent moniker that wraps an interface pointer to an object loaded in memory. Whereas most monikers identify objects that can be saved to persistent storage, pointer monikers identify objects that cannot. The associated creation function is **CreatePointerMoniker**.

A moniker provider must also implement other interfaces to allow the monikers it hands out to be bound. OLE objects that commonly provide monikers are link sources. These include server applications that support linking and container applications that support linking to their embedded objects.

Binding to an object means that a client uses a moniker to locate the object, activate it when necessary, and get a pointer to one of the active object's interfaces. The client of the moniker does not need to be aware of the class of the moniker – it must just get a pointer to the correct moniker's **IMoniker** interface. Monikers are used most often in this way by container applications that allow their documents to contain linked objects. However, link containers rarely call **IMoniker** methods directly. Instead, they generally manipulate linked objects through the default handler's implementation of the **IOleLink** interface, which calls the appropriate **IMoniker** methods as needed.

- **Class monikers** – these represent an object class. Class monikers bind to the class object of the class for which they are created. The associated creation function is **CreateClassComposite**.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPersist Methods	**Description**
GetClassID	Returns the object's CLSID.

IPersistStream Methods	**Description**
IsDirty	Checks whether object has been modified.
Load	Loads the object from a stream.
Save	Saves the object to a stream.
GetSizeMax	Returns the buffer size needed to save

the object.

IMoniker Methods	Description
BindToObject	Binds to the object named by the moniker.
BindToStorage	Binds to the object's storage.
Reduce	Reduces the moniker to simplest form.
ComposeWith	Composes with another moniker.
Enum	Enumerates component monikers.
IsEqual	Compares with another moniker.
Hash	Returns a hash value.
IsRunning	Checks whether object is running.
GetTimeOfLastChange	Returns time the object was last changed.
Inverse	Returns the inverse of the moniker.
CommonPrefixWith	Finds the prefix that the moniker has in common with another moniker.
RelativePathTo	Constructs a relative moniker between the moniker and another.
GetDisplayName	Returns the display name.
ParseDisplayName	Converts a display name into a moniker.
IsSystemMoniker	Checks whether moniker is one of the system-supplied types.

See Also

BindMoniker, CreateBindCtx, CreateGenericComposite, CreateFileMoniker, CreateItemMoniker, CreateAntiMoniker, CreatePointerMoniker, IOleLink, IPersistStream, IROTData, IMoniker—AntiMoniker Implementation, IMoniker—File Moniker Implementation, IMoniker—Item Moniker Implementation, IMoniker—Generic Composite Moniker Implementation, IMoniker—Pointer Moniker Implementation

IMoniker::BindToObject

Quick Info

Uses the moniker to bind to the object it identifies. The binding process involves finding the object, putting it into the running state if necessary, and supplying the caller with a pointer to a specified interface on the identified object.

```
HRESULT BindToObject(
    IBindCtx *pbc,          //Pointer to bind context object to be used
    IMoniker *pmkToLeft,    //Pointer to moniker that precedes this one in the composite
    REFIID riidResult,      //IID of interface pointer requested
    void **ppvResult        //Indirect pointer to the specified interface on the object
);
```

Parameters

pbc

[in] Pointer to the **IBindCtx** interface on the bind context object, which is used in this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the moniker implementation should retrieve information about its environment.

pmkToLeft

[in] If the moniker is part of a composite moniker, pointer to the moniker to the left of this moniker. This parameter is primarily used by moniker implementers to enable cooperation between the various components of a composite moniker. Moniker clients should pass NULL.

riidResult

[in] IID of the interface the client wishes to use to communicate with the object that the moniker identifies.

ppvResult

[out] When successful, indirect pointer to the interface specified in *riidResult* on the object the moniker identifies. In this case, the implementation must call **IUnknown::AddRef** on this pointer. It is the caller's responsibility to release the object with a call to **IUnknown::Release**. If an error occurs, *ppvResult* should return NULL.

Return Values

The method supports the standard return values E_UNEXPECTED and E_OUTOFMEMORY, as well as the following:

S_OK

The binding operation was successful.

MK_E_NOOBJECT

The object identified by this moniker, or some object identified by the composite moniker of which this moniker is a part, could not be found.

MK_E_EXCEEDEDDEADLINE

The binding operation could not be completed within the time limit specified by the bind context's **BIND_OPTS** structure.

MK_E_CONNECTMANUALLY

The binding operation requires assistance from the end user. The most common reasons for returning this value are that a password is needed or that a floppy needs to be mounted. When this value is returned, retrieve the moniker that caused the error with a call to **IBindCtx::GetObjectParam** with the key "ConnectManually". You can then call **IMoniker::GetDisplayName** to get the display name, display a dialog box that communicates the desired information, such as instructions to mount a floppy or a request for a password, and then retry the binding operation.

MK_E_INTERMEDIATEINTERFACENOTSUPPORTED

An intermediate object was found but it did not support an interface required to complete the binding operation. For example, an item moniker returns this value if its container does not support the **IOleItemContainer** interface.

STG_E_ACCESSDENIED

Unable to access the storage object.

IOleItemContainer::GetObject errors

If the moniker used to bind to an object contains an item moniker, errors associated with this method can be returned.

Remarks

IMoniker::BindToObject implements the primary function of a moniker, which is to locate the object identified by the moniker and return a pointer to one of its interfaces.

Notes to Callers

If you are using a moniker as a persistent connection between two objects, you activate the connection by calling **IMoniker::BindToObject**.

You typically call **IMoniker::BindToObject** during the following process:

1. Create a bind context object with a call to the **CreateBindCtx** function.
2. Call **IMoniker::BindToObject** using the moniker, retrieving a pointer to a desired interface on the identified object.
3. Release the bind context.
4. Through the acquired interface pointer, perform the desired operations on the object.
5. When finished with the object, release the object's interface pointer.

The following code fragment illustrates these steps:

```
// pMnk is an IMoniker * that points to a previously acquired moniker
// ICellRange is a custom interface designed for an object that is a
//            range of spreadsheet cells
ICellRange *pCellRange;
IBindCtx *pbc;

CreateBindCtx( 0, &pbc );
pMnk->BindToObject( pbc, NULL, IID_ICellRange, &pCellRange );
pbc->Release();
// pCellRange now points to the object; safe to use pCellRange
pCellRange->Release();
```

You can also use the **BindMoniker** function when you only intend one binding operation and don't need

to retain the bind context object. This helper function encapsulates the creation of the bind context, calling **IMoniker::BindToObject**, and releasing the bind context.

OLE containers that support links to objects use monikers to locate and get access to the linked object, but typically do not call **IMoniker::BindToObject** directly. Instead, when a user activates a link in a container, the link container usually calls **IOleObject::DoVerb**, using the link handler's implementation, which calls **IMoniker::BindToObject** on the moniker stored in the linked object (if it cannot handle the verb).

Notes to Implementers

What your implementation does depends on whether you expect your moniker to have a prefix, that is, whether you expect the *pmkToLeft* parameter to be NULL or not. For example, an item moniker, which identifies an object within a container, expects that *pmkToLeft* identifies the container. An item moniker consequently uses *pmkToLeft* to request services from that container. If you expect your moniker to have a prefix, you should use the *pmkToLeft* parameter (for instance, calling **IMoniker::BindToObject** on it) to request services from the object it identifies.

If you expect your moniker to have no prefix, your **IMoniker::BindToObject** implementation should first check the Running Object Table (ROT) to see if the object is already running. To acquire a pointer to the ROT, your implementation should call **IBindCtx::GetRunningObjectTable** on the *pbc* parameter. You can then call the **IRunningObjectTable::GetObject** method to see if the current moniker has been registered in the ROT. If so, you can immediately call **IUnknown::QueryInterface** to get a pointer to the interface requested by the caller.

When your **IMoniker::BindToObject** implementation binds to some object, it should use the *pbc* parameter to call **IBindCtx::RegisterObjectBound** to store a reference to the bound object in the bind context. This ensures that the bound object remains running until the bind context is released, which can avoid the expense of having a subsequent binding operation load it again later.

If the bind context's **BIND_OPTS** structure specifies the **BINDFLAGS_JUSTTESTEXISTENCE** flag, your implementation has the option of returning NULL in *ppvResult* (although you can also ignore the flag and perform the complete binding operation).

See Also

BindMoniker, **IMoniker::BindToStorage**

IMoniker::BindToStorage Quick Info

Retrieves an interface pointer to the storage that contains the object identified by the moniker. Unlike the **IMoniker::BindToObject** method, this method does not activate the object identified by the moniker.

```
HRESULT BindToStorage(
    IBindCtx *pbc,          //Pointer to bind context to be used
    IMoniker *pmkToLeft,    //Pointer to moniker to the left of this one in the composite
    REFIID riid,            //Reference to the identifier of the storage interface requested
    void **ppvObj           //Indirect pointer to interface on storage object containing the identified
                            object
);
```

Parameters

pbc

[in] Pointer to the **IBindCtx** interface on the bind context object to be used during this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the moniker implementation should retrieve information about its environment. For more information, see **IBindCtx**.

pmkToLeft

[in] If the moniker is part of a composite moniker, pointer to the moniker to the left of this moniker. This parameter is primarily used by moniker implementers to enable cooperation between the various components of a composite moniker. Moniker clients should pass NULL.

riid

[in] Reference to the identifier of the storage interface requested, whose pointer will be returned in *ppvObj*. Storage interfaces commonly requested include **IStorage**, **IStream**, and **ILockBytes**.

ppvObj

[out] Pointer to the interface identified by *riid* on the storage of the object identified by the moniker. If *ppvObj* is non-NULL, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs, *ppvObj* is set to NULL.

Return Values

The method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The binding operation was successful.

MK_E_NOSTORAGE

The object identified by this moniker does not have its own storage.

MK_E_EXCEEDEDDEADLINE

The operation could not be completed within the time limit specified by the bind context's **BIND_OPTS** structure.

MK_E_CONNECTMANUALLY

The operation was unable to connect to the storage, possibly because a network device could not be connected to. For more information, see **IMoniker::BindToObject**.

MK_E_INTERMEDIATEINTERFACENOTSUPPORTED

An intermediate object was found but it did not support an interface required for an operation. For example, an item moniker returns this value if its container does not support the **IOleItemContainer** interface.

STG_E_ACCESSDENIED

Unable to access the storage object.

IOleItemContainer::GetObject errors

Binding to a moniker containing an item moniker can return any of the errors associated with this function.

Remarks

There is an important difference between the **IMoniker::BindToObject** and **IMoniker::BindToStorage** methods. If, for example, you have a moniker that identifies a spreadsheet object, calling **IMoniker::BindToObject** provides access to the spreadsheet object itself, while calling **IMoniker::BindToStorage** provides access to the storage object in which the spreadsheet resides.

Notes to Callers

Although none of the OLE moniker classes call this method in their binding operations, it might be appropriate to call it in the implementation of a new moniker class. You could call this method in an implementation of **IMoniker::BindToObject** that requires information from the object identified by the *pmkToLeft* parameter and can get it from the persistent storage of the object without activation. For example, if your monikers are used to identify objects that can be activated without activating their containers, you may find this method useful.

A client that can read the storage of the object its moniker identifies could also call this method.

Notes to Implementers

Your implementation should locate the persistent storage for the object identified by the current moniker and return the desired interface pointer. Some types of monikers represent pseudo-objects, which are objects that do not have their own persistent storage. Such objects comprise some portion of the internal state of its container; as, for example, a range of cells in a spreadsheet. If your moniker class identifies this type of object, your implementation of **IMoniker::BindToStorage** should return the error MK_E_NOSTORAGE.

If the bind context's **BIND_OPTS** structure specifies the **BINDFLAGS_JUSTTESTEXISTENCE** flag, your implementation has the option of returning NULL in *ppvObj* (although it can also ignore the flag and perform the complete binding operation).

See Also

IMoniker::BindToObject

IMoniker::CommonPrefixWith Quick Info

Creates a new moniker based on the common prefix that this moniker (the one comprising the data of this moniker object) shares with another moniker.

```
HRESULT CommonPrefixWith(
    IMoniker *pmkOther,      //Pointer to moniker to be used for comparison
    IMoniker **ppmkPrefix    //Indirect pointer to the prefix
);
```

Parameters

pmkOther

[in] Pointer to the **IMoniker** interface on another moniker to be compared with this one to determine whether there is a common prefix.

ppmkPrefix

[out] When successful, points to the **IMoniker** pointer to the moniker that is the common prefix of this moniker and *pmkOther*. In this case, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs or if there is no common prefix, the implementation should set *ppmkPrefix* to NULL.

Return Values

The method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

A common prefix exists that is neither this moniker nor *pmkOther*.

MK_S_NOPREFIX

No common prefix exists.

MK_S_HIM

The entire *pmkOther* moniker is a prefix of this moniker.

MK_S_US

The two monikers are identical.

MK_S_ME

This moniker is a prefix of the *pmkOther* moniker.

MK_E_NOTBINDABLE

This method was called on a relative moniker. It is not meaningful to take the common prefix on a relative moniker.

Remarks

IMoniker::CommonPrefixWith creates a new moniker that consists of the common prefixes of the moniker on this moniker object and another moniker. If, for example, one moniker represents the path "c:\projects\secret\art\pict1.bmp" and another moniker represents the path "c:\projects\secret\docs\chap1.txt," the common prefix of these two monikers would be a moniker representing the path "c:\projects\secret."

Notes to Callers

The **IMoniker::CommonPrefixWith** method is primarily called in the implementation of the **IMoniker::RelativePathTo** method. Clients using a moniker to locate an object rarely need to call this method.

Call this method only if *pmkOther* and this moniker are both absolute monikers. An absolute moniker is either a file moniker or a generic composite whose leftmost component is a file moniker that represents an absolute path. Do not call this method on relative monikers, because it would not produce meaningful results.

Notes to Implementers

Your implementation should first determine whether *pmkOther* is a moniker of a class that you recognize and for which you can provide special handling (for example, if it is of the same class as this moniker). If so, your implementation should determine the common prefix of the two monikers. Otherwise, it should pass both monikers in a call to the **MonikerCommonPrefixWith** function, which correctly handles the generic case.

See Also

IMoniker::RelativePathTo, **MonikerCommonPrefixWith**

IMoniker::ComposeWith Quick Info

Combines the current moniker with another moniker, creating a new composite moniker.

```
HRESULT ComposeWith(
  IMoniker *pmkRight,          //Pointer to moniker to be composed onto this one
  BOOL fOnlyIfNotGeneric,      //Indicates if generic composition permissible
  IMoniker **ppmkComposite     //Indirect pointer to the composite
);
```

Parameters

pmkRight

[in] Pointer to the **IMoniker** interface on the moniker to compose onto the end of this moniker.

fOnlyIfNotGeneric

[in] If TRUE, the caller requires a non-generic composition, so the operation should proceed only if *pmkRight* is a moniker class that this moniker can compose with in some way other than forming a generic composite. If FALSE, the method can create a generic composite if necessary.

ppmkComposite

[out] When the call is successful, indirect pointer to the location of the resulting composite moniker pointer. In this case, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs or if the monikers compose to nothing (e.g., composing an anti-moniker with an item moniker or a file moniker), *ppmkComposite* should be set to NULL.

Return Values

The method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The monikers were successfully combined.

MK_E_NEEDGENERIC

Indicates that *fOnlyIfNotGeneric* was TRUE, but the monikers could not be composed together without creating a generic composite moniker.

Remarks

Joining two monikers together is called composition. Sometimes two monikers of the same class can be combined in what is called non-generic composition. For example, a file moniker representing an incomplete path and another file moniker representing a relative path can be combined to form a single file moniker representing the complete path. Non-generic composition for a given moniker class can be handled only in the implementation of **IMoniker::ComposeWith** for that moniker class.

Combining two monikers of any class is called generic composition, which can be accomplished through a call to the **CreateGenericComposite** function.

Composition of monikers is an associative operation. That is, if A, B, and C are monikers, then, where

Comp() represents the composition operation:

Comp(Comp(A, B), C)

is always equal to

Comp(A, Comp(B, C))

Notes to Callers

To combine two monikers, you should call **IMoniker::ComposeWith** rather than calling the **CreateGenericComposite** function to give the first moniker a chance to perform a non-generic composition.

An object that provides item monkers to identify its objects would call **IMoniker::ComposeWith** to provide a moniker that completely identifies the location of the object. This would apply, for example, to a server that supports linking to portions of a document, or a container that supports linking to embedded objects within its documents. In such a situation, you would do the following:

1. Create an item moniker identifying an object.
2. Get a moniker that identifies the object's container.
3. Call **IMoniker::ComposeWith** on the moniker identifying the container, passing the item moniker as the *pmkRight* parameter.

Most callers of **IMoniker::ComposeWith** should set the *fOnlyIfNotGeneric* parameter to FALSE.

Notes to Implementers

You can use either non-generic or generic composition to compose the current moniker with the moniker that *pmkRight* points to. If the class of the moniker indicated by *pmkRight* is the same as that of the current moniker, it is possible to use the contents of *pmkRight* to perform a more intelligent non-generic composition.

In writing a new moniker class, you must decide if there are any kinds of monikers, whether of your own class or another class, to which you want to give special treatment. If so, implement **IMoniker::ComposeWith** to check whether *pmkRight* is a moniker of the type that should have this treatment. To do this, you can call the moniker's **GetClassID** method (derived from the **IPersist** Interface), or, if you have defined a moniker object that supports a custom interface, you can call **IUnknown::QueryInterface** on the moniker for that interface. An example of special treatment would be the non-generic composition of an absolute file moniker with a relative file moniker. The most common case of a special moniker is the inverse for your moniker class (whatever you return from your implementation of **IMoniker::Inverse**).

If *pmkRight* completely negates the receiver so the resulting composite is empty, you should pass back **NULL** in *ppmkComposite* and return the status code S_OK.

If the *pmkRight* parameter is not of a class to which you give special treatment, examine *fOnlyIfNotGeneric* to determine what to do next. If *fOnlyIfNotGeneric* is TRUE, pass back NULL through *ppmkComposite* and return the status code MK_E_NEEDGENERIC. If *fOnlyIfNotGeneric* is FALSE, call the **CreateGenericComposite** function to perform the composition generically.

See Also

CreateGenericComposite, **IMoniker::Inverse**

IMoniker::Enum Quick Info

Supplies a pointer to an enumerator that can enumerate the components of a composite moniker.

```
HRESULT Enum(
  BOOL fForward,                  //Specifies direction of enumeration
  IEnumMoniker **ppenumMoniker    //Indirect pointer to the IEnumMoniker pointer
);
```

Parameters

fForward

[in] If TRUE, enumerates the monikers from left to right. If FALSE, enumerates from right to left.

ppenumMoniker

[out] When successful, indirect pointer to an **IEnumMoniker** enumerator on this moniker. In this case, the implementation must call **IUnknown::AddRef** on the parameter. It is the caller's responsibility to call **IUnknown::Release**. If an error occurs or if the moniker has no enumerable components, the implementation sets *ppenumMoniker* to NULL.

Return Values

The method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

Indicates success. This value is returned even if the moniker does not provide an enumerator (if *ppenumMoniker* equals NULL).

Remarks

IMoniker::Enum must supply an **IEnumMoniker** pointer to an enumerator that can enumerate the components of a moniker. For example, the implementation of the **IMoniker::Enum** method for a generic composite moniker creates an enumerator that can determine the individual monikers that make up the composite, while the **IMoniker::Enum** method for a file moniker creates an enumerator that returns monikers representing each of the components in the path.

Notes to Callers

Call this method to examine the components that make up a composite moniker.

Notes to Implementers

If the new moniker class has no discernible internal structure, your implementation of this method can simply return S_OK and set *ppenumMoniker* to NULL.

See Also

IEnum*XXXX*

IMoniker::GetDisplayName Quick Info

Gets the display name , which is a user-readable representation of this moniker.

```
HRESULT GetDisplayName(
  IBindCtx *pbc,                //Pointer to bind context to be used
  IMoniker *pmkToLeft,          //Pointer to moniker to the left in the composite
  LPOLESTR *ppszDisplayName     //Indirect pointer to the display name
);
```

Parameters

pbc

[in] Pointer to the **IBindCtx** interface on the bind context to be used in this operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the moniker implementation should retrieve information about its environment. For more information, see **IBindCtx**.

pmkToLeft

[in] If the moniker is part of a composite moniker, pointer to the moniker to the left of this moniker. This parameter is primarily used by moniker implementers to enable cooperation between the various components of a composite moniker. Moniker clients should pass NULL.

ppszDisplayName

[out] When successful, indirect pointer to a zero-terminated wide character string (two bytes per character) containing the display name of this moniker. The implementation must use **IMalloc::Alloc** to allocate the string returned in *ppszDisplayName*, and the caller is responsible for calling **IMalloc::Free** to free it. Both the caller and and the one called use the OLE task allocator returned by **CoGetMalloc**. If an error occurs, *ppszDisplayName* should be set to NULL.

Return Values

The method supports the standard return value E_OUTOFMEMORY

, as well as the following:

S_OK

The display name was successfully supplied.

MK_E_EXCEEDEDDEADLINE

The binding operation could not be completed within the time limit specified by the bind context's **BIND_OPTS** structure.

E_NOTIMPL

There is no display name.

Remarks

IMoniker::GetDisplayName provides a string that is a displayable representation of the moniker. A display name is not a complete representation of a moniker's internal state; it is simply a form that can be read by users. As a result, it is possible (though rare) for two different monikers to have the same display

name. While there is no guarantee that the display name of a moniker can be parsed back into that moniker when calling the **MkParseDisplayName** function with it, failure to do so is rare.

As examples, the file moniker implementation of this method supplies the path the moniker represents, and an item moniker's display name is the string identifying the item that is contained in the moniker.

Notes to Callers

It is possible that retrieving a moniker's display name may be an expensive operation. For efficiency, you may want to cache the results of the first successful call to **IMoniker::GetDisplayName**, rather than making repeated calls.

Notes to Implementers

If you are writing a moniker class in which the display name does not change, simply cache the display name and supply the cached name when requested. If the display name can change over time, getting the current display name might mean that the moniker has to access the object's storage or bind to the object, either of which can be expensive operations. If this is the case, your implementation of **IMoniker::GetDisplayName** should return MK_E_EXCEEDEDDEADLINE if the name cannot be retrieved by the time specified in the bind context's **BIND_OPTS** structure.

A moniker that is intended to be part of a generic composite moniker should include any preceding delimiter (such as '\') as part of its display name. For example, the display name returned by an item moniker includes the delimiter specified when it was created with the **CreateItemMoniker** function. The display name for a file moniker does not include a delimiter because file monikers are always expected to be the leftmost component of a composite.

See Also

IMoniker::ParseDisplayName, **MkParseDisplayName**

IMoniker::GetTimeOfLastChange Quick Info

Provides a number representing the time the object identified by this moniker was last changed. To be precise, the time returned is the earliest time OLE can identify after which no change has occurred, so this time may be later than the time of the last change to the object.

```
HRESULT GetTimeOfLastChange(
    IBindCtx *pbc,          //Bind context to be used
    IMoniker *pmkToLeft,    //Moniker to the left in the composite
    FILETIME *pFileTime     //Receives the time of last change
);
```

Parameters

pbc

[in] Pointer to the bind context to be used in this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the moniker implementation should retrieve information about its environment. For more information, see **IBindCtx**.

pmkToLeft

[in] If the moniker is part of a composite moniker, pointer to the moniker to the left of this moniker. This parameter is primarily used by moniker Implementers to enable cooperation between the various components of a composite moniker. Moniker clients should pass NULL.

pFileTime

[out] Pointer to the **FILETIME** structure receiving the time of last change. A value of {0xFFFFFFFF,0x7FFFFFFF} indicates an error (for example, exceeded time limit, information not available).

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The method successfully returned a time.

MK_E_EXCEEDEDDEADLINE

The binding operation could not be completed within the time limit specified by the bind context's **BIND_OPTS** structure.

MK_E_CONNECTMANUALLY

The operation was unable to connect to the storage for this object, possibly because a network device could not be connected to. For more information, see **IMoniker::BindToObject**.

MK_E_UNAVAILABLE

The time of the change is unavailable, and will not be available no matter what deadline is used.

Remarks

Notes to Callers

If you're caching information returned by the object identified by the moniker, you may want to ensure that your information is up-to-date. To do so, you would call **IMoniker::GetTimeOfLastChange** and compare the time returned with the time you last retrieved information from the object.

For the monikers stored within linked objects, **IMoniker::GetTimeOfLastChange** is primarily called by the default handler's implementation of **IOleObject::IsUpToDate**. Container applications call **IOleObject::IsUpToDate** to determine if a linked object (or an embedded object containing linked objects) is up-to-date without actually binding to the object. This enables an application to determine quickly which linked objects require updating when the end user opens a document. The application can then bind only those linked objects that need updating (after prompting the end user to determine whether they should be updated), instead of binding every linked object in the document.

Notes to Implementers

It is important to perform this operation quickly because, for linked objects, this method is called when a user first opens a compound document. Consequently, your **IMoniker::GetTimeOfLastChange** implementation should not bind to any objects. In addition, your implementation should check the deadline parameter in the bind context and return MK_E_EXCEEDEDDEADLINE if the operation cannot be completed by the specified time.

There are a number of strategies you can use in your implementations:

- For many types of monikers, the *pmkToLeft* parameter identifies the container of the object identified by this moniker. If this is true of your moniker class, you can simply call **IMoniker::GetTimeOfLastChange** on the *pmkToLeft* parameter, since an object cannot have changed at a date later than its container.
- You can get a pointer to the Running Object Table (ROT) by calling **IBindCtx::GetRunningObjectTable** on the *pbc* parameter, and then calling **IRunningObjectTable::GetTimeOfLastChange**, since the ROT generally records the time of last change.
- You can get the storage associated with this moniker (or the *pmkToLeft* moniker) and return the storage's last modification time with a call to **IStorage::Stat**.

See Also

IBindCtx::GetRunningObjectTable, **IRunningObjectTable::GetTimeOfLastChange**

IMoniker::Hash

Quick Info

Calculates a 32-bit integer using the internal state of the moniker.

```
HRESULT Hash(
  DWORD *pdwHash      //Pointer to hash value
);
```

Parameter

pdwHash

[out] Pointer to the hash value.

Return Value

S_OK

Successfully received a 32-bit integer hash value.

Remarks

Notes to Callers

You can use the value returned by this method to maintain a hash table of monikers. The hash value determines a hash bucket in the table. To search such a table for a specified moniker, calculate its hash value and then compare it to the monikers in that hash bucket using **IMoniker::IsEqual**.

Notes to Implementers

The hash value must be constant for the lifetime of the moniker. Two monikers that compare as equal using **IMoniker::IsEqual** must hash to the same value.

Marshaling and then unmarshaling a moniker should have no effect on its hash value. Consequently, your implementation of **IMoniker::Hash** should rely only on the internal state of the moniker, not on its memory address.

See Also

IMoniker::IsEqual

IMoniker::Inverse

Quick Info

Provides a moniker that, when composed to the right of this moniker or one of similar structure, will destroy it (the moniker will compose to nothing).

```
HRESULT Inverse(

  IMoniker **ppmk      //Indirect pointer to the inverse of the moniker
);
```

Parameter

ppmk

[out] When successful, indirect pointer to the **IMoniker** interface on a moniker that is the inverse of this moniker. In this case, the implementation must call **IUnknown::AddRef** on the parameter. It is the caller's responsibility to call **IUnknown::Release**. If an error occurs, the implementation should set *ppmk* to NULL.

Return Values

The method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The inverse moniker has been returned successfully.

MK_E_NOINVERSE

The moniker class does not have an inverse.

Remarks

The inverse of a moniker is analogous to the ".." directory in MS-DOS file systems; the ".." directory acts as the inverse to any other directory name, because appending ".." to a directory name results in an empty path. In the same way, the inverse of a moniker typically is also the inverse of all monikers in the same class. However, it is not necessarily the inverse of a moniker of a different class.

The inverse of a composite moniker is a composite consisting of the inverses of the components of the original moniker, arranged in reverse order. For example, if the inverse of A is Inv(A) and the composite of A, B, and C is Comp(A, B, C), then

Inv(Comp(A, B, C))

is equal to

Comp(Inv(C), Inv(B), Inv(A)).

Not all monikers have inverses. Most monikers that are themselves inverses, such as anti-monikers, do not have inverses. Monikers that have no inverse cannot have relative monikers formed from inside the objects they identify to other objects outside.

Notes to Callers

An object that is using a moniker to locate another object usually does not know the class of the moniker it is using. To get the inverse of a moniker, you should always call **IMoniker::Inverse** rather than the **CreateAntiMoniker** function, because you cannot be certain that the moniker you're using considers an

anti-moniker to be its inverse.

The **IMoniker::Inverse** method is also called by the implementation of the **IMoniker::RelativePathTo** method, to assist in constructing a relative moniker.

Notes to Implementers

If your monikers have no internal structure, you can call the **CreateAntiMoniker** function in to get an anti-moniker in your implementation of **IMoniker::Inverse**. In your implementation of **IMoniker::ComposeWith**, you need to check for the inverse you supply in the implementation of **IMoniker::Inverse**.

See Also

CreateAntiMoniker, **IMoniker::ComposeWith**, **IMoniker::RelativePathTo**

IMoniker::IsEqual Quick Info

Compares this moniker with a specified moniker and indicates whether they are identical.

```
HRESULT IsEqual(

  IMoniker *pmkOtherMoniker      //Pointer to moniker to be used for comparison
);
```

Parameter

pmkOtherMoniker

[in] Pointer to the **IMoniker** interface on the moniker to be used for comparison with this one (the one from which this method is called).

Return Values

S_OK

The two monikers are identical.

S_FALSE

The two monikers are not identical.

Remarks

Previous implementations of the Running Object Table (ROT) called this method. The current implementation of the ROT uses the **IROTData** interface instead.

Notes to Callers

Call this method to determine if two monikers are identical or not. Note that the reduced form of a moniker is considered different from the unreduced form. You should call the **IMoniker::Reduce** method before calling **IMoniker::IsEqual**, because a reduced moniker is in its most specific form. **IMoniker::IsEqual** may return S_FALSE on two monikers before they are reduced, and S_OK after they are reduced.

Notes to Implementers

Your implementation should not reduce the current moniker before performing the comparison. It is the caller's responsibility to call **IMoniker::Reduce** in order to compare reduced monikers.

Note that two monikers that compare as equal must hash to the same value using **IMoniker::Hash**.

See Also

IMoniker::Reduce, **IMoniker::Hash**, **IROTData**

IMoniker::IsRunning Quick Info

Determines whether the object identified by this moniker is currently loaded and running.

```
HRESULT IsRunning(
  IBindCtx *pbc,                //Pointer to bind context to be used
  IMoniker *pmkToLeft,          //Pointer to moniker to the left in the composite
  IMoniker *pmkNewlyRunning     //Pointer to moniker of a newly running object
);
```

Parameters

pbc

[in] Pointer to the**IBindCtx** interface on the bind context to be used in this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the moniker implementation should retrieve information about its environment. For more information, see **IBindCtx**.

pmkToLeft

[in] Pointer to the**IMoniker** interface on the moniker to the left of this moniker if this moniker is part of a composite. This parameter is primarily used by moniker Implementers to enable cooperation between the various components of a composite moniker; moniker clients can usually pass NULL.

pmkNewlyRunning

[in] Pointer to the**IMoniker** interface on the moniker most recently added to the Running Object Table (ROT). This can be NULL. If non-NULL, the implementation can return the results of calling **IMoniker::IsEqual** on the *pmkNewlyRunning* parameter, passing the current moniker. This parameter is intended to enable **IMoniker::IsRunning** implementations that are more efficient than just searching the ROT, but the implementation can choose to ignore *pmkNewlyRunning* without causing any harm.

Return Values

The method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The moniker is running.

S_FALSE

The moniker is not running.

Remarks

Notes to Callers

If speed is important when you're requesting services from the object identified by the moniker, you may want those services *only* if the object is already running (because loading an object into the running state may be time-consuming). In such a situation, you'd call **IMoniker::IsRunning** to determine if the object is running.

For the monikers stored within linked objects, **IMoniker::IsRunning** is primarily called by the default

handler's implementation of **IOleLink::BindIfRunning**.

Notes to Implementers

To get a pointer to the Running Object Table (ROT), your implementation should call **IBindCtx::GetRunningObjectTable** on the *pbc* parameter. Your implementation can then call **IRunningObjectTable::IsRunning** to determine whether the object identified by the moniker is running. Note that the object identified by the moniker must have registered itself with the ROT when it first began running.

See Also

IOleLink::BindIfRunning, **IBindCtx::GetRunningObjectTable**, **IRunningObjectTable::IsRunning**

IMoniker::IsSystemMoniker Quick Info

Indicates whether this moniker is of one of the system-supplied moniker classes.

```
HRESULT IsSystemMoniker(
  DWORD *pdwMksys      //Pointer to value from MKSYS enumeration
);
```

Parameter

pdwMksys

[out] Pointer to an integer that is one of the values from the **MKSYS** enumeration, and refers to one of the OLE moniker classes. This parameter cannot be NULL.

Return Values

S_OK

The moniker is a system moniker.

S_FALSE

The moniker is not a system moniker.

Remarks

Notes to Callers

New values of the **MKSYS** enumeration may be defined in the future; therefore you should explicitly test for each value you are interested in.

Notes to Implementers

Your implementation of this method must return **MKSYS_NONE**. You cannot use this function to identify your own monikers (for example, in your implementation of **IMoniker::ComposeWith**). Instead, you should use your moniker's implementation of **IPersist::GetClassID** or use **IUnknown::QueryInterface** to test for your own private interface.

See Also

IPersist::GetClassID, **MKSYS**

IMoniker::ParseDisplayName Quick Info

Reads as many characters of the specified display name as it understands and builds a moniker corresponding to the portion read; this procedure is known as "parsing" the display name.

```
HRESULT ParseDisplayName(
  IBindCtx *pbc,                //Pointer to bind context to be used
  IMoniker *pmkToLeft,          //Pointer to moniker to the left in the composite
  LPOLESTR pszDisplayName,      //Pointer to display name
  ULONG *pchEaten,              //Pointer to number of characters consumed
  IMoniker **ppmkOut            //Indirect pointer to moniker built from display name
);
```

Parameters

pbc

[in] Pointer to the **IBindCtx** interface on the bind context to be used in this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the moniker implementation should retrieve information about its environment. For more information, see **IBindCtx**.

pmkToLeft

[in] Pointer to the **IMoniker** interface on the moniker that has been built out of the display name up to this point.

pszDisplayName

[in] Pointer to a zero-terminated string containing the remaining display name to be parsed. For Win32 applications, the **LPOLESTR** type indicates a wide character string (two bytes per character); otherwise, the string has one byte per character.

pchEaten

[out] Pointer to the number of characters in *pszDisplayName* that were consumed in this step.

ppmkOut

[out] When successful, indirect pointer to the **IMoniker** interface on the moniker that was built from *pszDisplayName*. In this case, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs, the implementation sets *ppmkOut* to NULL.

Return Values

The method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The parsing operation was completed successfully.

MK_E_SYNTAX

An error in the syntax of the input components (*pmkToLeft*, this moniker, and *pszDisplayName*). For example, a file moniker returns this error if *pmkToLeft* is non-NULL, and an item moniker returns it if *pmkToLeft* is NULL.

IMoniker::BindToObject errors

Parsing display names may cause binding. Thus, any error associated with this function may be returned.

Remarks

Notes to Callers

Moniker clients do not typically call **IMoniker::ParseDisplayName** directly. Instead, they call the **MkParseDisplayName** function when they want to convert a display name into a moniker (for example, in implementing the Links dialog box for a container application, or for implementing a macro language that supports references to objects outside the document). That function first parses the initial portion of the display name itself.

It then calls **IMoniker::ParseDisplayName** on the moniker it has just created, passing the remainder of the display name and getting a new moniker in return; this step is repeated until the entire display name has been parsed.

Notes to Implementers

Your implementation may be able to perform this parsing by itself if your moniker class is designed to designate only certain kinds of objects. Otherwise, you must get an **IParseDisplayName** interface pointer for the object identified by the moniker-so-far (i.e., the composition of *pmkToLeft* and this moniker) and then return the results of calling **IParseDisplayName::ParseDisplayName**.

There are different strategies for getting an **IParseDisplayName** pointer:

- You can try to get the object's CLSID (by calling **IPersist::GetClassID** on the object), and then call the **CoGetClassObject** function, requesting the **IParseDisplayName** interface on the class factory associated with that CLSID.
- You can try to bind to the object itself to get an **IParseDisplayName** pointer.
- You can try binding to the object identified by *pmkToLeft* to get an **IOleItemContainer** pointer, and then call **IOleItemContainer::GetObject** to get an **IParseDisplayName** pointer for the item.

Any objects that are bound should be registered with the bind context (see **IBindCtx::RegisterObjectBound**) to ensure that they remain running for the duration of the parsing operation.

See Also

IParseDisplayName, **MkParseDisplayName**

IMoniker::Reduce

Quick Info

Returns a reduced moniker; that is, another moniker that refers to the same object as this moniker but can be bound with equal or greater efficiency.

```
HRESULT Reduce(
  IBindCtx *pbc,                //Pointer to bind context to be used
  DWORD dwReduceHowFar,         //How much reduction should be done
  IMoniker **ppmkToLeft,        //Indirect pointer to moniker to the left in the composite
  IMoniker **ppmkReduced        //Indirect pointer to the reduced moniker
);
```

Parameters

pbc

[in] Pointer to the **IBindCtx** interface on the bind context to be used in this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the moniker implementation should retrieve information about its environment. For more information, see **IBindCtx**.

dwReduceHowFar

[in] DWORD that specifies how far this moniker should be reduced. This parameter must be one of the values from the **MKRREDUCE** enumeration.

ppmkToLeft

[in, out] On entry, indirect pointer to the moniker to the left of this moniker, if this moniker is part of a composite. This parameter is primarily used by moniker Implementers to enable cooperation between the various components of a composite moniker; moniker clients can usually pass NULL.

On return, *ppmkToLeft* is usually set to NULL, indicating no change in the original moniker to the left. In rare situations *ppmkToLeft* indicates a moniker, indicating that the previous moniker to the left should be disregarded and the moniker returned through *ppmkToLeft* is the replacement. In such a situation, the implementation must call **IUnknown::Release** on the passed-in pointer and call **IUnknown::AddRef** on the returned moniker; the caller must release it later. If an error occurs, the implementation can either leave the parameter unchanged or set it to NULL.

ppmkReduced

[out] Indirect pointer to the **IMoniker** interface on the reduced form of this moniker, which can be NULL if an error occurs or if this moniker is reduced to nothing. If this moniker cannot be reduced, *ppmkReduced* is simply set to this moniker and the return value is MK_S_REDUCED_TO_SELF. If *ppmkReduced* is non-NULL, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. (This is true even if *ppmkReduced* is set to this moniker.)

Return Values

The method supports the standard return values E_UNEXPECTED and E_OUTOFMEMORY, as well as the following:

S_OK

This moniker was reduced.

MK_S_REDUCED_TO_SELF

This moniker could not be reduced any further, so *ppmkReduced* indicates this moniker.

MK_E_EXCEEDEDDEADLINE

The operation could not be completed within the time limit specified by the bind context's **BIND_OPTS** structure.

Remarks

IMoniker::Reduce is intended for the following uses:

- It enables the construction of user-defined macros or aliases as new kinds of moniker classes. When reduced, the moniker to which the macro evaluates is returned.
- It enables the construction of a kind of moniker that tracks data as it moves about. When reduced, the moniker of the data in its current location is returned.
- On file systems that support an identifier-based method of accessing files which is independent of file names; a file moniker could be reduced to a moniker which contains one of these identifiers.

The intent of the **MKRREDUCE** flags passed in the *dwReduceHowFar* parameter is to provide the ability to programmatically reduce a moniker to a form whose display name is recognizable to the user. For example, paths in the file system, bookmarks in word-processing documents, and range names in spreadsheets are all recognizable to users. In contrast, a macro or an alias encapsulated in a moniker are not recognizable to users.

Notes to Callers

The scenarios described above are not currently implemented by the system-supplied moniker classes.

You should call **IMoniker::Reduce** before comparing two monikers using the **IMoniker::IsEqual** method, because a reduced moniker is in its most specific form. **IMoniker::IsEqual** may return S_FALSE on two monikers before they are reduced and return S_OK after they are reduced.

Notes to Implementers

If the current moniker can be reduced, your implementation must not reduce the moniker in-place. Instead, it must return a new moniker that represents the reduced state of the current one. This way, the caller still has the option of using the non-reduced moniker (for example, enumerating its components). Your implementation should reduce the moniker *at least* as far as is requested.

See Also

IMoniker::IsEqual, **MKRREDUCE**

IMoniker::RelativePathTo Quick Info

Supplies a moniker that, when composed onto the end of this moniker (or one with a similar structure), yields the specified moniker.

```
HRESULT RelativePathTo(
  IMoniker *pmkOther,        //Pointer to moniker to which a relative path should be taken
  IMoniker **ppmkRelPath     //Indirect pointer to the relative moniker
);
```

Parameters

pmkOther

[in] Pointer to the **IMoniker** interface on the moniker to which a relative path should be taken.

ppmkRelPath

[out] Iindirect pointer to the **IMoniker** interface on the relative moniker. When successful, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs, the implementation sets *ppmkRelPath* to NULL.

Return Values

The method supports the standard return values E_OUTOFMEMORY and

E_UNEXPECTED, as well as the following:

S_OK

A meaningful relative path has been returned.

MK_S_HIM

No common prefix is shared by the two monikers and the moniker returned in *ppmkRelPath* is *pmkOther*.

MK_E_NOTBINDABLE

This moniker is a relative moniker, such as an item moniker. This moniker must be composed with the moniker of its container before a relative path can be determined.

Remarks

A relative moniker is analogous to a relative path (such as "..\backup"). For example, suppose you have one moniker that represents the path "c:\projects\secret\art\pict1.bmp" and another moniker that represents the path "c:\projects\secret\docs\chap1.txt." Calling **IMoniker::RelativePathTo** on the first moniker, passing the second one as the *pmkOther* parameter, would create a relative moniker representing the path "..\docs\chap1.txt."

Notes to Callers

Moniker clients typically do not need to call **IMoniker::RelativePathTo**. This method is primarily called by the default handler for linked objects. Linked objects contain both an absolute and a relative moniker to identify the link source (this enables link tracking if the user moves a directory tree containing both the container and source files). The default handler calls this method to create a relative moniker from the container document to the link source (that is, it calls **IMoniker::RelativePathTo** on the moniker

identifying the container document, passing the moniker identifying the link source as the *pmkOther* parameter).

If you do call **IMoniker::RelativePathTo**, call it only on absolute monikers; for example, a file moniker or a composite moniker whose leftmost component is a file moniker, where the file moniker represents an absolute path. Do not call this method on relative monikers.

Notes to Implementers

Your implementation of **IMoniker::RelativePathTo** should first determine whether *pmkOther* is a moniker of a class that you recognize and for which you can provide special handling (for example, if it is of the same class as this moniker). If so, your implementation should determine the relative path. Otherwise, it should pass both monikers in a call to the **MonikerRelativePathTo** function, which correctly handles the generic case.

The first step in determining a relative path is determining the common prefix of this moniker and *pmkOther*. The next step is to break this moniker and *pmkOther* into two parts each, say (P, myTail) and (P, otherTail) respectively, where P is the common prefix. The correct relative path is then the inverse of myTail composed with otherTail:

Comp(Inv(myTail), otherTail)

Where Comp() represents the composition operation and Inv() represents the inverse operation.

Note that for certain types of monikers, you cannot use your **IMoniker::Inverse** method to construct the inverse of myTail. For example, a file moniker returns an anti-moniker as an inverse, while its **IMoniker::RelativePathTo** method must use one or more file monikers that each represent the path ".." to construct the inverse of myTail.

See Also

IMoniker::Inverse, **IMoniker::CommonPrefixWith**, **MonikerRelativePathTo**

IMoniker - Anti-Moniker Implementation

Anti-monikers are the inverse of the OLE implementations of file, item, and pointer monikers. That is, an anti-moniker composed to the right of a file moniker, item moniker, or pointer moniker composes to nothing.

When To Use

If you're a moniker client, you typically do not need to use anti-monikers. When you need the inverse of a moniker, you should call **IMoniker::Inverse**. For example, if you need an inverse to remove the last piece of a composite moniker, use **IMoniker::Enum** to enumerate the pieces of the moniker and call **IMoniker::Inverse** on the rightmost piece. You shouldn't use an anti-moniker for this purpose because you can't be sure that the rightmost piece of a composite considers an anti-moniker to be its inverse.

The only situation in which you should explicitly use an anti-moniker is if you are writing a new moniker class and if you have no special requirements for constructing inverses to your monikers. In that situation, you can return anti-monikers from your implementation of **IMoniker::Inverse**. In your implementation of **IMoniker::ComposeWith**, you should then annihilate one of your monikers for every anti-moniker you encounter.

Remarks

IMoniker::BindToObject

This method is not implemented. It returns E_NOTIMPL.

IMoniker::BindToStorage

This method is not implemented. It returns E_NOTIMPL.

IMoniker::Reduce

This method returns MK_S_REDUCED_TO_SELF and passes back the same moniker.

IMoniker::ComposeWith

If *fOnlyIfNotGeneric* is TRUE, this method sets *ppmkComposite* to NULL moniker and returns MK_E_NEEDGENERIC; otherwise, the method returns the result of combining the two monikers into a generic composite. Note that composing a file, item, or pointer moniker to the right of an anti-moniker produces a generic composite rather than composing to nothing, as would be the case if the order of composition were reversed.

IMoniker::Enum

This method returns S_OK and sets **ppenumMoniker* to NULL.

IMoniker::IsEqual

This method returns S_OK if both are anti-monikers; otherwise, it returns S_FALSE.

IMoniker::Hash

This method calculates a hash value for the moniker.

IMoniker::IsRunning

This method checks the ROT to see if the object is running.

IMoniker::GetTimeOfLastChange

This method is not implemented (that is, it returns E_NOTIMPL).

IMoniker::Inverse

This method returns MK_E_NOINVERSE and sets **ppmk* to NULL.

IMoniker::CommonPrefixWith

If the other moniker is also an anti-moniker, the method returns MK_S_US and sets *ppmkPrefix* to this moniker. Otherwise, the method calls the **MonikerCommonPrefixWith** function. This function correctly handles the case where the other moniker is a generic composite.

IMoniker::RelativePathTo

This method returns MK_S_HIM and sets **ppmkRelPath* to the other moniker.

IMoniker::GetDisplayName

For each anti-moniker contained in this moniker, this method return one instance of "\.."

IMoniker::ParseDisplayName

This method is not implemented (that is, it returns E_NOTIMPL).

IMoniker::IsSystemMoniker

This method returns S_OK and indicates MKSYS_ANTIMONIKER.

See Also

CreateAntiMoniker, **IMoniker**

IMoniker - Class Moniker Implementation

Class monikers are monikers that represent an object class. Class monikers bind to the class object of the class for which they are created.

Class monikers are most useful in composition with other types of monikers, such as file monikers or item monikers. Class monikers may also be composed to the right of monikers supporting binding to the **IClassActivator** interface. This allows **IClassActivator** to provide access to the class object and instances of the class.

When to Use

To use class monikers, you must use the **CreateClassMoniker** function to create the monikers.

Remarks

IMoniker::BindToObject

If *pmkLeft* is NULL, calls **CoGetClassObject** using the CLSID the class moniker was initialized with (in **CreateClassMoniker** or through **MkParseDisplayName**) and the CLSCTX of the current pbc (IBindContext).

If *pmkLeft* is non-NULL, calls *pmkLeft*->**BindToObject** for **IClassActivator** and calls **IClassActivator::GetClassObject** with the CLSID it was initialized with and the CLSCTX and LOCALE parameters from of the current *pbc* (**IBindContext**).

This process is very roughly sketched out in the following code:

```
BIND_OPTS2    bindOpts;
IClassActivator *pActivate;

bindOpts.cbStruct = sizeof(bindOpts);
pbc->GetBindOptions(&bindOpts);

if (NULL == pmkToLeft)
    return CoGetClassObject(<clsid>, bindOpts.dwClassContext, NULL,
riid, ppvResult);

    pmkToLeft->BindToObject(pbc, NULL, IID_IClassActivator, (void **)
&pActivate);
    hr = pActivate->GetClassObject(<clsid>, bindOpts.dwClassContext,
bindOpts.locale, iid, ppvResult);
    pActivate->Release();
    return hr;
```

IMoniker::BindToStorage

This method forwards to the class moniker's **BindToObject**.

IMoniker::Reduce

This method returns MK_S_REDUCED_TO_SELF and passes back the same moniker.

IMoniker::ComposeWith

Follows the contract, and behaves like an Item Moniker in that it can return E_INVALIDARG and MK_E_NEEDGENERIC, etc.

IMoniker::Enum

This method returns S_OK and sets *ppenumMoniker* to NULL. May return E_INVALIDARG if

ppenumMoniker is an invalid pointer.

IMoniker::IsEqual

This method returns S_OK if *pmkOther* is a class moniker constructed with the same CLSID information as itself. Otherwise, the method returns S_FALSE. May return E_INVALIDARG if *pmkOther* is an invalid pointer.

IMoniker::Hash

This method calculates a hash value for the moniker and returns S_OK. may return E_INVALIDARG if *pdwHash* is an invalid pointer.

IMoniker::IsRunning

Returns E_NOTIMPL.

IMoniker::GetTimeOfLastChange

Returns MK_E_UNAVAILABLE.

IMoniker::Inverse

This method returns an anti-moniker (i.e., the results of calling **CreateAntiMoniker**).

IMoniker::CommonPrefixWith

If *pmkOther* **IsEqual** to this moniker, retrives a pointer to this moniker and returns MK_S_US. If *pmkOther* is a class moniker but is not equal to this moniker, returns MK_E_NOPREFIX. Otherwise returns the result of calling **MonikerCommonPrefixWith** with itself as *pmkThis*, *pmkOther* and *ppmkPrefix*, which handles the case whre *pmkOther* is a generic composite moniker.

IMoniker::RelativePathTo

This method returns the result of calling This method returns the result of calling **MonikerRelativePathTo** with *pmkSrc* equal to this moniker, *pmkOther*, *ppmkRelPath*, and TRUE as *dwReserved*.

IMoniker::GetDisplayName

The display name for class monikers is of the form:

display-name = "CLSID:" string-clsid-no-curly-braces *[";" clsid-options] ":"

clsid-options = clsid-param "=" value

clsid-param = none currently defined

Example:

clsid:a7b90590-36fd-11cf-857d-00aa006d2ea4:

IMoniker::ParseDisplayName

This method parses the display name by binding to itself for IParseDisplayName and asking the bound object to parse the display name into a moniker, as follows:

```
    hr = BindToObject(pbc, pmkToLeft, IID_IParseDisplayName,
(void**)&ppdn);
    if (SUCCEEDED(hr)) {
        hr = ppdn->ParseDisplayName(pbc, lpszDisplayName, pchEaten,
ppmkOut);
        ppdn->Release();
        }
    return hr;
```

This method tries to acquire an **IParseDisplayName** pointer, first by binding to the class factory for the object identified by the moniker, and then by binding to the object itself. If either of these binding operations is successful, the file moniker passes the unparsed portion of the display name to the

IParseDisplayName::ParseDisplayName method.

This method returns MK_E_SYNTAX if *pmkToLeft* is non-NULL.

IMoniker::IsSystemMoniker

This method returns S_OK, and passes back MKSYS_CLASSMONIKER.

See Also

CreateClassMoniker, **IMoniker**

IMoniker - File Moniker Implementation

File monikers are monikers that represent a path in the file system; a file moniker can identify any object that is saved in its own file. To identify objects contained within a file, you can compose monikers of other classes (for example, item monikers) to the right of a file moniker. However, the moniker to the left of a file moniker within a composite must be another file moniker, an anti-moniker, or a class moniker. It is illegal, for example, for an item moniker to appear to the left of a file moniker in a composite.

Note that an anti-moniker is the inverse of an entire file moniker, not the inverse of a component of the path that the moniker represents; that is, when you compose an anti-moniker to the right of a file moniker, the entire file moniker is removed. If you want to remove just the rightmost component of the path represented by a file moniker, you must create a separate file moniker based on the ".." path and then compose that to the end of the file moniker.

When to Use

If you're a moniker client (that is, you're using a moniker to get an interface pointer to an object), you typically don't need to know the class of the moniker you're using; you simply call methods using an **IMoniker** interface pointer.

If you're a moniker provider (that is, you're handing out monikers that identify your objects to make them accessible to moniker clients), you must use file monikers if the objects you're identifying are stored in files. If each object resides in its own file, file monikers are the only type you need. If the objects you're identifying are smaller than a file, you need to use another type of moniker (for example, item monikers) in addition to file monikers.

To use file monikers, you must use the **CreateFileMoniker** function to create the monikers. In order to allow your objects to be loaded when a file moniker is bound, your objects must implement the **IPersistFile** interface.

The most common example of moniker providers are OLE server applications that support linking. If your OLE server application supports linking only to file-based documents in their entirety, file monikers are the only type of moniker you need. If your OLE server application supports linking to objects smaller than a document (such as sections of a document or embedded objects), you must use item monikers as well as file monikers.

Remarks

IMoniker::BindToObject

When *pmkToLeft* is NULL, the method looks for the moniker in the ROT, and if found, queries the retrieved object for the requested interface pointer. If the moniker is not found in the ROT, the method loads the object from the file system and retrieves the requested interface pointer.

If *pmkLeft* is not NULL, then instead of determining the class to instantiate and initialize with the contents of the file referred to by the file moniker using **GetClassFile** (or other means), call pmkLeft->BindToObject for **IClassFactory** and **IClassActivator**, retrieve this pointer in *pcf*.

If this fails with E_NOINTERFACE, return MK_E_INTERMEDIATEINTERFACENOTSUPPORTED.

If the **IClassFactory** pointer is successfully retrieved, call pcf->CreateInstance(IID_IPersistFile, (void**)&ppf) to get a fresh instance of the class to be initialized and initialize it using **IPersistFile** or other appropriate means per the existing initialization paths of File moniker.

IMoniker::BindToStorage

This method opens the file specified by the path represented by the moniker and returns an **IStorage** pointer to that file. The method supports binding to **IStorage** interface only; if **IStream** or **ILockBytes** is requested in *riid*, the method returns E_UNSPEC, and if other interfaces are requested, this method returns E_NOINTERFACE. **IStream** and **ILockBytes** will be supported in future releases.

Unless *pmkToLeft* is a class moniker, *pmkToLeft* should be NULL, as in the implementation of **IMoniker::BindToObject**. For situations where *pmkToLeft* is non-NULL, see the above description.

IMoniker::Reduce

This method returns MK_S_REDUCED_TO_SELF and passes back the same moniker.

IMoniker::ComposeWith

If *pmkRight* is an anti-moniker, the returned moniker is NULL. If *pmkRight* is a composite whose leftmost component is an anti-moniker, the returned moniker is the composite with the leftmost anti-moniker removed. If *pmkRight* is a file moniker, this method collapses the two monikers into a single file moniker, if possible. If not possible (e.g., if both file monikers represent absolute paths, as in *d:\work* and *e:\reports*), then the returned moniker is NULL and the return value is MK_E_SYNTAX. If *pmkRight* is neither an anti-moniker nor a file moniker, then the method checks the *fOnlyIfNotGeneric* parameter; if it is FALSE, the method combines the two monikers into a generic composite; if it is TRUE, the method sets **ppmkComposite* to NULL and returns MK_E_NEEDGENERIC.

IMoniker::Enum

This method returns S_OK and sets *ppenumMoniker* to NULL.

IMoniker::IsEqual

This method returns S_OK if **pmkOther* is a file moniker and the paths for both monikers are identical (using a case-insensitive comparison). Otherwise, the method returns S_FALSE.

IMoniker::Hash

This method calculates a hash value for the moniker.

IMoniker::IsRunning

If *pmkNewlyRunning* is non-NULL, this method returns TRUE if that moniker is equal to this moniker. Otherwise, the method asks the ROT whether this moniker is running. The method ignores *pmkToLeft*.

IMoniker::GetTimeOfLastChange

If this moniker is in the ROT, this method returns the last change time registered there; otherwise, it returns the last write time for the file. If the file cannot be found, this method returns MK_E_NOOBJECT.

IMoniker::Inverse

This method returns an anti-moniker (i.e., the results of calling **CreateAntiMoniker**).

IMoniker::CommonPrefixWith

If both monikers are file monikers, this method returns a file moniker that is based on the common components at the beginning of two file monikers. Components of a file moniker can be:

- A machine name of the form *server**share*. A machine name is treated as a single component, so two monikers representing the paths "\\myserver\public\work" and "\\myserver\private\games" do not have "\\myserver" as a common prefix.
- A drive designation (for example, "C:").
- A directory or file name.

If the other moniker is not a file moniker, this method passes both monikers in a call to the **MonikerCommonPrefixWith** function. This function correctly handles the case where the other moniker is a generic composite.

This method returns MK_E_NOPREFIX if there is no common prefix.

IMoniker::RelativePathTo

This method computes a moniker which when composed to the right of this moniker yields the other moniker. For example, if the path of this moniker is "C:\work\docs\report.doc" and if the other moniker is "C:\work\art\picture.bmp," then the path of the computed moniker would be "..\..\art\picture.bmp."

IMoniker::GetDisplayName

This method returns the path that the moniker represents. If this method is called by a 16-bit application, the method checks to see whether the specified file exists and, if so, returns a short name for that file because 16-bit applications are not equipped to handle long file names.

IMoniker::ParseDisplayName

This method tries to acquire an **IParseDisplayName** pointer, first by binding to the class factory for the object identified by the moniker, and then by binding to the object itself. If either of these binding operations is successful, the file moniker passes the unparsed portion of the display name to the **IParseDisplayName::ParseDisplayName** method.

This method returns MK_E_SYNTAX if *pmkToLeft* is non-NULL.

IMoniker::IsSystemMoniker

This method returns S_OK, and passes back MKSYS_FILEMONIKER.

See Also

CreateFileMoniker, **IMoniker**, **IPersistFile**

IMoniker - Generic Composite Moniker Implementation

A generic composite moniker is a composite moniker whose components have no special knowledge of each other.

Composition is the process of joining two monikers together. Sometimes two monikers of specific classes can be combined in a special manner; for example, a file moniker representing an incomplete path and another file moniker representing a relative path can be combined to form a single file moniker representing the complete path. This is an example of "non-generic" composition. "Generic" composition, on the other hand, can connect any two monikers, no matter what their classes. Because a non-generic composition depends on the class of the monikers involved, it can be performed only by a particular class's implementation of the **IMoniker::ComposeWith** method. You can define new types of non-generic compositions if you write a new moniker class. By contrast, generic compositions are performed by the **CreateGenericComposite** function.

When to Use

If you're a moniker client (that is, you're using a moniker to get an interface pointer to an object), you typically don't need to know the class of the moniker you're using, or whether it is a generic composite or a non-generic composite; you simply call methods using an **IMoniker** interface pointer.

If you're a moniker provider (that is, you're handing out monikers that identify your objects to make them accessible to moniker clients), you may have to compose two monikers together. (For example, if you are using an item moniker to identify an object, you must compose it with the moniker identifying the object's container before you hand it out.) You use the **IMoniker::ComposeWith** method to do this, calling the method on the first moniker and passing the second moniker as a parameter; this method may produce either a generic or a non-generic composite.

The only time you should explicitly create a generic composite moniker is if you are writing your own moniker class. In your implementation of **IMoniker::ComposeWith**, you should attempt to perform a non-generic composition whenever possible; if you cannot perform a non-generic composition and generic composition is acceptable, you can call the **CreateGenericComposite** function to create a generic composite moniker.

Remarks

IMoniker::BindToObject

If *pmkToLeft* is NULL, this method looks for the moniker in the ROT, and if found, queries the retrieved object for the requested interface pointer. If *pmkToLeft* is not NULL, the method recursively calls **IMoniker::BindToObject** on the rightmost component of the composite, passing the rest of the composite as the *pmkToLeft* parameter for that call.

IMoniker::BindToStorage

This method recursively calls **BindToStorage** on the rightmost component of the composite, passing the rest of the composite as the *pmkToLeft* parameter for that call.

IMoniker::Reduce

This method recursively calls **Reduce** for each of its component monikers. If any of the components reduces itself, the method returns S_OK and passes back a composite of the reduced components. If no reduction occurred, the method passes back the same moniker and returns MK_S_REDUCED_TO_SELF.

IMoniker::ComposeWith

If *fOnlyIfNotGeneric* is TRUE, this method sets **pmkComposite* to NULL and returns MK_E_NEEDGENERIC; otherwise, the method returns the result of combining the two monikers by

calling the **CreateGenericComposite** function.

IMoniker::Enum

If successful, this method returns S_OK and passes back an enumerator that enumerates the component monikers that make up the composite; otherwise, the method returns E_OUTOFMEMORY.

IMoniker::IsEqual

This method returns S_OK if the components of both monikers are equal when compared in the left-to-right order.

IMoniker::Hash

This method calculates a hash value for the moniker.

IMoniker::IsRunning

If *pmkToLeft* is non-NULL, this method composes *pmkToLeft* with this moniker and calls **IsRunning** on the result.

If *pmkToLeft* is NULL, this method returns TRUE if *pmkNewlyRunning* is non-NULL and is equal to this moniker.

If *pmkToLeft* and *pmkNewlyRunning* are both NULL, this method checks the ROT to see whether the moniker is running. If so, the method returns S_OK; otherwise, it recursively calls **IMoniker::IsRunning** on the rightmost component of the composite, passing the remainder of the composite as the *pmkToLeft* parameter for that call. This handles the case where the moniker identifies a pseudo-object that is not registered as running; see the Item Moniker implementation of **IMoniker::IsRunning**.

IMoniker::GetTimeOfLastChange

This method creates a composite of *pmkToLeft* (if non-NULL) and this moniker and uses the ROT to retrieve the time of last change. If the object is not in the ROT, the method recursively calls **IMoniker::GetTimeOfLastChange** on the rightmost component of the composite, passing the remainder of the composite as the *pmkToLeft* parameter for that call.

IMoniker::Inverse

This method returns a composite moniker that consists of the inverses of each of the components of the original composite, stored in reverse order. For example, if the inverse of *A* is $A^{(-1)}$, then the inverse of the composite $A \circ B \circ C$ is $C^{(-1)} \circ B^{(-1)} \circ A^{(-1)}$.

IMoniker::CommonPrefixWith

If the other moniker is a composite, this method compares the components of each composite from left to right. The returned common prefix moniker might also be a composite moniker, depending on how many of the leftmost components were common to both monikers. If the other moniker is not a composite, the method simply compares it to the leftmost component of this moniker.

If the monikers are equal, the method returns MK_S_US and sets *ppmkPrefix* to this moniker. If the other moniker is a prefix of this moniker, the method returns MK_S_HIM and sets *ppmkPrefix* to the other moniker. If this moniker is a prefix of the other, this method returns MK_S_ME and sets *ppmkPrefix* to this moniker.

If there is no common prefix, this method returns MK_E_NOPREFIX and sets *ppmkPrefix* to NULL.

IMoniker::RelativePathTo

This method finds the common prefix of the two monikers and creates two monikers that consist of the remainder when the common prefix is removed. Then it creates the inverse for the remainder of this moniker and composes the remainder of the other moniker on the right of it.

IMoniker::GetDisplayName

This method returns the concatenation of the display names returned by each component moniker of the composite.

IMoniker::ParseDisplayName

This method recursively calls **IMoniker::ParseDisplayName** on the rightmost component of the composite, passing everything else as the *pmkToLeft* parameter for that call.

IMoniker::IsSystemMoniker

This method returns S_OK and indicates MKSYS_GENERICCOMPOSITE.

See Also

CreateGenericComposite, **IMoniker**

IMoniker - Item Moniker Implementation

Item monikers are used to identify objects within containers, such as a portion of a document, an embedded object within a compound document, or a range of cells within a spreadsheet. Item monikers are often used in combination with file monikers; a file moniker is used to identify the container while an item moniker is used to identify the item within the container.

An item moniker contains a text string; this string is used by the container object to distinguish the contained item from the others. The container object must implement the **IOleItemContainer** interface; this interface enables the item moniker code to acquire a pointer to an object, given only the string that identifies the object.

When to Use

If you're a moniker client (that is, you're using a moniker to get an interface pointer to an object), you typically don't need to know the class of the moniker you're using; you simply call methods using an **IMoniker** interface pointer.

If you're a moniker provider (that is, you're handing out monikers that identify your objects to make them accessible to moniker clients), you must use item monikers if the objects you're identifying are contained within another object and can be individually identified using a string. You'll also need to use another type of moniker (for example, file monikers) in order to identify the container object.

To use item monikers, you must use the **CreateItemMoniker** function to create the monikers. In order to allow your objects to be loaded when an item moniker is bound, the container of your objects must implement the **IOleItemContainer** interface.

The most common example of moniker providers are OLE applications that support linking. If your OLE application supports linking to objects smaller than a file-based document, you need to use item monikers. For a server application that allows linking to a selection within a document, you use the item monikers to identify those objects. For a container application that allows linking to embedded objects, you use the item monikers to identify the embedded objects.

Remarks

IMoniker::BindToObject

If *pmkToLeft* is NULL, this method returns E_INVALIDARG. Otherwise, the method calls **IMoniker::BindToObject** on the *pmkToLeft* parameter, requesting an **IOleItemContainer** interface pointer. The method then calls **IOleItemContainer::GetObject**, passing the string contained within the moniker, and returns the requested interface pointer.

IMoniker::BindToStorage

If *pmkToLeft* is NULL, this method returns E_INVALIDARG. Otherwise, the method calls **IMoniker::BindToObject** on the *pmkToLeft* parameter, requesting an **IOleItemContainer** interface pointer. The method then calls **IOleItemContainer::GetObjectStorage** for the requested interface.

IMoniker::Reduce

This method returns MK_S_REDUCED_TO_SELF and passes back the same moniker.

IMoniker::ComposeWith

If *pmkRight* is an anti-moniker, the returned moniker is NULL; if *pmkRight* is a composite whose leftmost component is an anti-moniker, the returned moniker is the composite after the leftmost anti-moniker is removed. If *pmkRight* is not an anti-moniker, the method combines the two monikers into a generic composite if *fOnlyIfNotGeneric* is FALSE; if *fOnlyIfNotGeneric* is TRUE, the method returns a NULL moniker and a return value of MK_E_NEEDGENERIC.

IMoniker::Enum

This method returns S_OK and sets **ppenumMoniker* to NULL.

IMoniker::IsEqual

This method returns S_OK if both monikers are item monikers and their display names are identical (using a case-insensitive comparison); otherwise, the method returns S_FALSE.

IMoniker::Hash

This method calculates a hash value for the moniker.

IMoniker::IsRunning

If *pmkToLeft* is NULL, this method returns TRUE if *pmkNewlyRunning* is non-NULL and is equal to this moniker. Otherwise, the method checks the ROT to see whether this moniker is running.

If *pmkToLeft* is non-NULL, the method calls **IMoniker::BindToObject** on the *pmkToLeft* parameter, requesting an **IOleItemContainer** interface pointer. The method then calls **IOleItemContainer::IsRunning**, passing the string contained within this moniker.

IMoniker::GetTimeOfLastChange

If *pmkToLeft* is NULL, this method returns MK_E_NOTBINDABLE. Otherwise, the method creates a composite of *pmkToLeft* and this moniker and uses the ROT to access the time of last change. If the object is not in the ROT, the method calls **IMoniker::GetTimeOfLastChange** on the *pmkToLeft* parameter.

IMoniker::Inverse

This method returns an anti-moniker (i.e., the results of calling **CreateAntiMoniker**).

IMoniker::CommonPrefixWith

If the other moniker is an item moniker that is equal to this moniker, this method sets **ppmkPrefix* to this moniker and returns MK_S_US; otherwise, the method calls the **MonikerCommonPrefixWith** function. This function correctly handles the case where the other moniker is a generic composite.

IMoniker::RelativePathTo

This method returns MK_E_NOTBINDABLE and sets **ppmkRelPath* to NULL.

IMoniker::GetDisplayName

This method returns the concatenation of the delimiter and the item name that were specified when the item moniker was created.

IMoniker::ParseDisplayName

If *pmkToLeft* is NULL, this method returns MK_E_SYNTAX. Otherwise, the method calls **IMoniker::BindToObject** on the *pmkToLeft* parameter, requesting an **IOleItemContainer** interface pointer. The method then calls **IOleItemContainer::GetObject**, requesting an **IParseDisplayName** interface pointer to the object identified by the moniker, and passes the display name to **IParseDisplayName::ParseDisplayName**.

IMoniker::IsSystemMoniker

This method returns S_OK and indicates MKSYS_ITEMMONIKER.

See Also

CreateItemMoniker, **IMoniker**, **IOleItemContainer**

IMoniker - Pointer Moniker Implementation

A pointer moniker essentially wraps an interface pointer so that it looks like a moniker and can be passed to those interfaces that require monikers. Binding a pointer moniker is done by calling the pointer's **QueryInterface** method.

Instances of pointer monikers refuse to be serialized, that is, **IPersistStream::Save** will return an error. These monikers can, however, be marshaled to a different process in an RPC call; internally, the system marshals and unmarshals the pointer using the standard paradigm for marshaling interface pointers.

When to Use

Pointer monikers are rarely needed. Use pointer monikers only if you need monikers to identify objects that have no persistent representation. Pointer monikers allow such objects to participate in a moniker-binding operation.

Remarks

IMoniker::BindToObject

This method queries the wrapped pointer for the requested interface.

IMoniker::BindToStorage

This method queries the wrapped pointer for the requested interface.

IMoniker::Reduce

This method returns MK_S_REDUCED_TO_SELF and passes back the same moniker.

IMoniker::ComposeWith

If *pmkRight* is an anti-moniker, the returned moniker is NULL; if *pmkRight* is a composite whose leftmost component is an anti-moniker, the returned moniker is the composite after the leftmost anti-moniker is removed. If *fOnlyIfNotGeneric* is FALSE, the returned moniker is a generic composite of the two monikers; otherwise, the method sets **ppmkComposite* to NULL and returns MK_E_NEEDGENERIC.

IMoniker::Enum

This method is not implemented (that is, it returns E_NOTIMPL).

IMoniker::IsEqual

This method returns S_OK only if both are pointer monikers and the interface pointers that they wrap are identical.

IMoniker::Hash

This method calculates a hash value for the moniker.

IMoniker::IsRunning

This method always returns S_OK, because the object identified by a pointer moniker must always be running.

IMoniker::GetTimeOfLastChange

This method is not implemented (that is, it returns E_NOTIMPL).

IMoniker::Inverse

This method returns an anti-moniker (i.e., the results of calling **CreateAntiMoniker**).

IMoniker::CommonPrefixWith

If the two monikers are equal, this method returns MK_S_US and sets **ppmkPrefix* to this moniker. Otherwise, the method returns MK_E_NOPREFIX and sets **ppmkPrefix* to NULL.

IMoniker::RelativePathTo

This method is not implemented (that is, it returns E_NOTIMPL).

IMoniker::GetDisplayName

This method is not implemented (that is, it returns E_NOTIMPL).

IMoniker::ParseDisplayName

This method queries the wrapped pointer for the **IParseDisplayName** interface and passes the display name to **IParseDisplayName::ParseDisplayName**.

IMoniker::IsSystemMoniker

This method returns S_OK and indicates MKSYS_POINTERMONIKER.

See Also

IMoniker, **CreatePointerMoniker**

IMultiQI Quick Info

The **IMultiQI** interface enables a client to query an object proxy, or handler, for multiple interfaces, using a single RPC call. By using this interface, instead of relying on separate calls to **IUnknown::QueryInterface**, clients can reduce the number of RPC calls that have to cross thread, process, or machine boundaries and, therefore, the amount of time required to obtain the requeseted interface pointers.

When to Implement

You never have to implement this interface because there is no situation in which it is required. OLE server applications that rely on COM's standard remoting support get the interface for free because COM implements it on every object proxy. The only situation in which you might want to implement this interface yourself is if you are writing a custom marshaler that handles interface remoting. Even here, implementing **IMultiQI** yourself is not recommended, particularly if your object is aggregatable.

When to Use

When more than one interface pointer is sought, client applications should **QueryInterface** for **IMultiQI** and use it if available.

Methods in VTable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IMultiQI Methods	**Description**
QueryMultipleInterfaces	Queries for multiple interfaces.

See Also

IUnknown::QueryInterface

IMultiQI::QueryMultipleInterfaces Quick Info

Fills a caller-provided array of structures with pointers to multiple interfaces.Calling this method is equivalent to issuing a series of separate **QueryInterface** calls except that you do not incur the overhead of a corresponding number of RPC calls. In multithreaded applications and distributed environments, keeping RPC calls to a minimum is essential for optimal performance.

```
HRESULT QueryMultipleInterfaces(
    ULONG cMQIs         // Number of structures in array
    MULTI_QI *pMQIs     // Pointer to first structure in array
    );
```

Parameters

cMQIs

[in] Pointer to the number of elements in an array of **MULTI_QI** structures, each of which contains the IID of a single interface.

pMQIs

[in, out] Pointer to the first **MULTI_QI** strucutre in the array.

Return Value

S_OK

Pointers were returned to all requested interfaces.

S_FALSE

Pointers were returned to some, but not all, of the requested interfaces.

E_NOINTERFACE

Pointers were returned to none of the requested interfaces.

Remarks

The **QueryMultipleInterfaces** method takes as input an array of **MULTI_QI** structures. Each structure specifies an interface IID and contains two additional blank fields for receiving an interface pointer and return value.

This method obtains as many requested interface pointers as possible directly from the object proxy. For each interface not implemented on the proxy, the method calls the server to obtain a pointer. Upon receiving an interface pointer from the server, the method builds a corresponding interface proxy and returns its pointer along with pointers to the interfaces it already implements.

Notes to Callers

A caller should begin by querying the object proxy for the **IMultiQI** interface. If the object proxy returns a pointer to this interface, the caller should then create a **MULTI_QI** structure for each interface it wants to obtain. Each structure should specify an interface IID and set its *pItf* member to NULL. Failure to set the *pItf* member to NULL will cause the object proxy to ignore the structure.

On return, **QueryMultipleInterfaces** writes the requested interface pointer and a return value into each **MULTI_QI** structure in the client's array. The *pItf* field receives the pointer; the *hr* field receives the return

value.

If the value returned from a call to **QueryMultipleInterfaces** is S_OK, then pointers were returned for all requested interfaces. If the return value is E_NOINTERFACE, then pointers were returned for none of the requested interfaces. If the return value is S_FALSE, then pointers to one or more requested interfaces were not returned.In this event, the client should check the *hr* field of each MULTI_QI structure to determine which interfaces were acquired and which were not.

If a client knows ahead of time that it will be using several of an object's interfaces, it can call **QueryMultipleInterfaces** up front and then, later, if a **QueryInterface** is done for one of the interfaces already acquired through **QueryMultipleInterfaces**, no RPC call will be necessary.

On return, the caller should check the *hr* field of each **MULTI_QI** structure to determine which interface pointers were and were not returned.

The client is responsible for releasing each of the acquired interfaces by calling **IUnknown::Release**.

See Also

IUnknown

IObjectWithSite Quick Info

The **IObjectWithSite** interface provides a simple way to support communication between an object and its site in the container.

Often an object needs to communicate directly with a container site object and, in effect, manage the site object itself. Outside of **IOleObject::SetClientSite**, there is no generic means through which an object becomes aware of its site. **IObjectWithSite** provides simple objects with a simple siting mechanism (lighter than **IOleObject**) This interface should only be used when **IOleObject** is not already in use.

Through **IObjectWithSite**, a container can pass the **IUnknown** pointer of its site to the object through **IObjectWithSite::SetSite**. Callers can also retrieve the latest site passed to **IObjectWithSite::SetSite** through **IObjectWithSite::GetSite**. This latter method is included as a hooking mechanism, allowing a third party to intercept calls from the object to the site.

When to Implement

An object implements this interface so it's container can supply it with an interface pointer for its site object. Then, the object can communicate directly with its site.

When to Use

A container calls the **SetSite** method on this interface to provide an object with an interface pointer for its site.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IObjectWithSite Methods	Description
SetSite	Provides the site's **IUnknown** pointer to the object being managed.
GetSite	Retrieves the last site set with **IObjectWithSite::SetSite**.

IObjectWithSite::GetSite Quick Info

Retrieves the last site set with **IObjectWithSite::SetSite**. If there's no known site, the object return a failure code.

```
HRESULT GetSite(
  REFIID riid,        //IID of interface pointer being requested
  void** ppvSite      //Indirect pointer to caller's void
);
```

Parameters

riid

[in] The IID of the interface pointer that should be returned in *ppvSite*.

ppvSite

[out] Indirect pointer to the caller's **void** * variable in which the object stores the interface pointer of the site last seen in **IObjectWithSite::SetSite**. The specific interface returned depends on the *riid* argument—in essence, the two arguments act identically to those in **IUnknown::QueryInterface**. If the appropriate interface pointer is available, the object must call **IUnknown::AddRef** on that pointer before returning successfully. If no site is available, or the requested interface is not supported, the object sets this argument to NULL and returns a failure code.

Return Values

S_OK

The site was returned successfully and the caller is responsible for calling **IUnknown::Release** when the site is no longer needed.

E_FAIL

There is no site in which case **ppvSite* contains NULL on return.

E_NOINTERFACE

There is a site but it does not support the interface requested by *riid*.

Remarks

E_NOTIMPL is not allowed—any object implementing this interface must be able to return the last site seen in **IObjectWithSite::SetSite**.

IObjectWithSite::SetSite Quick Info

Provides the site's **IUnknown** pointer to the object. The object should hold onto this pointer, calling **IUnknown::AddRef** in doing so. If the object already has a site, it should call that existing site's **IUnknown::Release**, save the new site pointer, and call the new site's **IUnknown::AddRef**.

```
HRESULT SetSite(
  IUnknown* pUnkSite      //Pointer to IUnknown of the site managing this object
);
```

Parameter

pUnkSite

[in] Pointer to the **IUnknown** interface pointer of the site managing this object. If NULL, the object should call **IUnknown::Release** on any existing site at which point the object no longer knows its site.

Return Value

S_OK

Returned in all circumstances.

Remarks

E_NOTIMPL is not allowed—without implementation of the **SetSite** method, the **IObjectWithSite** interface is unnecessary.

See Also

IObjectWithSite::GetSite

IOleAdviseHolder Quick Info

The **IOleAdviseHolder** interface contains methods that manage advisory connections and compound document notifications in an object server. Its methods are intended to be used to implement the advisory methods of **IOleObject**. **IOleAdviseHolder** is implemented on an advise holder object. Its methods establish and delete advisory connections from the object managed by the server to the object's container, which must contain an advise sink (support the **IAdviseSink** interface). The advise holder object must also keep track of which advise sinks are interested in which notifications and pass along the notifications as appropriate.

When to Implement

It is unlikely that you would choose to implement this interface. OLE provides an implementation of the OLE advise holder as a convenience to programmers.Few applications will require notification capabilities beyond those which the default advise holder provides. In general, a single server application that requires different notification capabilities would simply implement the advisory functionality in its **IOleObject** advisory methods. It would be necessary to implement **IOleAdviseHolder** only in the case where there may be a need for a custom advise holder object, whose methods are to be used to implement the **IOleObject** methods in a set of servers.

When to Use

Call the methods of **IOleAdviseHolder** to implement the advisory methods of **IOleObject**. Applications instantiate an OLE advise holder by calling the OLE function **CreateOleAdviseHolder**. (OLE also provides a data advise holder to manage data notifications. Applications create a data advise holder by calling the OLE function **CreateDataAdviseHolder**.)

Containers and other objects that need to receive compound document notifications must implement the **IAdviseSink** interface to receive those notifications, and call the **IOleAdviseHolder** interface methods to establish an advisory connection and inform the object of what specific notifications it wishes to receive.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IOleAdviseHolder Methods	Description
Advise	Establishes advisory connection with sink.
Unadvise	Deletes advisory connection with sink.
EnumAdvise	Supplies an **IEnumSTATDATA** pointer to an enumeration object that can be used to determine current advisory connections.
SendOnRename	Advises sink that name of object has changed.
SendOnSave	Advises sink that object has been saved.
SendOnClose	Advises sink that object has been closed.

See Also

CreateOleAdviseHolder, IDataAdviseHolder, CreateDataAdviseHolder, IAdviseSink, IOleObject

IOleAdviseHolder::Advise Quick Info

Establishes an advisory connection between an OLE object and the calling object's advise sink. Through that sink, the calling object can receive notification when the OLE object is renamed, saved, or closed.

```
HRESULT Advise(
    IAdviseSink * pAdvise,          //Pointer to the advise sink on the calling object
    DWORD * pdwConnection           //Pointer to a token
);
```

Parameters

pAdvise

[in] Pointer to the **IAdviseSink** interface on the advisory sink that should be informed of changes.

pdwConnection

[out] Pointer to a token that can be passed to the **IOleAdviseHolder::Unadvise** method to delete the advisory connection. The calling object is responsible for calling both **IUnknown::AddRef** and **IUnknown::Release** on this pointer.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Advisory connections set up successfully.

Remarks

Containers, object handlers, and link objects all create advise sinks to receive notification of changes in compound-document objects of interest, such as embedded or linked objects. OLE objects of interest to these objects must implement the **IOleObject** interface, which includes several advisory methods, including **IOleObject::Advise**. A call to this method must set up an advisory connection with any advise sink that calls it, and maintain each connection until it is closed. It must be able to handle more than one advisory connection at a time.

IOleAdviseHolder::Advise is intended to be used to simplify the implementation of **IOleObject::Advise**. You can get a pointer to the OLE implementation of **IOleAdviseHolder** by calling **CreateOleAdviseHolder**, and then, to implement **IOleObject::Advise**, just delegate the call to **IOleAdviseHolder::Advise**. Other **IOleAdviseHolder** methods are intended to implement other **IOleObject** advisory methods.

If the attempt to establish an advisory connection is successful, the object receiving the call returns a nonzero value through *pdwConnection*. If the attempt fails, the object returns a zero. To delete an advisory connection, the object with the advise sink passes this nonzero token back to the object by calling **IOleAdviseHolder::Unadvise**.

See Also

IOleAdviseHolder::UnAdvise, **IOleAdviseHolder::EnumAdvise**, **IOleObject::Advise**

IOleAdviseHolder::EnumAdvise

Quick Info

Creates an enumerator that can be used to enumerate the advisory connections currently established for an object, and supplies a pointer to its **IEnumSTATDATA** interface.

```
HRESULT EnumAdvise(

  IEnumSTATDATA ** ppenumAdvise      //Indirect pointer to the new enumerator
);
```

Parameter

ppenumAdvise

[out] Indirect pointer to the **IEnumSTATDATA** interface on the new enumerator. A NULL value indicates that there are presently no advisory connections on the object, or that an error occured. The advise holder is responsible for incrementing the reference count on the **IEnumSTATDATA** pointer this method supplies. It is the caller's responsibility to call **IUnknown::Release** when it is done with the pointer.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

Enumerator created successfully.

E_NOTIMPL

EnumAdvise is not implemented.

Remarks

IOleAdviseHolder::EnumAdvise creates an enumerator that can be used to enumerate an object's established advisory connections. The method supplies a pointer to the **IEnumSTATDATA** interface on this enumerator. Advisory connection information for each connection is stored in the **STATDATA** structure, and the enumerator must be able to enumerate these structures, defined as follows:

```
typedef struct tagSTATDATA {
    FORMATETC         Formatetc;
    DWORD             grfAdvf;
    IAdviseSink *     pAdvise;
    DWORD             dwConnection;
    }STATDATA;
```

For this method, the only relevant structure members are *pAdvise* and *dwConnection*. Other members contain data advisory information. When you call the enumeration methods, and while an enumeration is in progress, the effect of registering or revoking advisory connections on what is to be enumerated is undefined.

See Also

IOleAdviseHolder::Advise, **IOleAdviseHolder::UnAdvise**, **IOleObject::EnumAdvise**, **IDataAdviseHolder::EnumAdvise**, **STATDATA** structure

IOleAdviseHolder::SendOnClose Quick Info

Sends notification that the object has closed to all advisory sinks currently registered with the advise holder.

HRESULT SendOnClose();

Return Value

S_OK

Advise sinks were notified of the close operation through a call to the **IAdviseSink::OnClose** method.

Remarks

IOleAdviseHolder::SendOnClose must call **IAdviseSink::OnClose** on all advise sinks that have a valid advisory connection with the object, whenever the object goes from the running state to the loaded state. This occurs through a call to **IOleObject::Close**, so you can call **IOleAdviseHolder::SendOnClose** when you determine that a Close operation has been successful.

See Also

IAdviseSink::OnClose

IOleAdviseHolder::SendOnRename Quick Info

Sends **IAdviseSink::OnRename** notifications to all advisory sinks currently registered with the advise holder.

```
HRESULT SendOnRename(

  IMoniker *pmk       //Pointer to an interface on the new moniker
);
```

Parameter

pmk

[in] Pointer to the new full moniker of the object.

Return Value

S_OK

Advise sinks were sent **IAdviseSink::OnRename** notifications.

Remarks

IOleAdviseHolder::SendOnRename calls **IAdviseSink::OnRename** to advise the calling object, which must have already established an advisory connection, that the object has a new moniker. If you are using the OLE advise holder (having obtained a pointer through a call to **CreateOleAdviseHolder**), you can call **IOleAdviseHolder::SendOnRename** in the implementation of **IOleObject::SetMoniker**, when you have determined that the operation is successful.

See Also

IAdviseSink::OnRename

IOleAdviseHolder::SendOnSave Quick Info

Sends **IAdviseSink::OnSave** notifications to all advisory sinks currently registered with the advise holder.

HRESULT SendOnSave();

Return Value

S_OK

Advise sinks were sent **IAdviseSink::OnSave** notifications.

Remarks

IOleAdviseHolder::SendOnSave calls **IAdviseSink::OnSave** to advise the calling object (client), which must have already established an advisory connection, that the object has been saved. If you are using the OLE advise holder (having obtained a pointer through a call to **CreateOleAdviseHolder**), you can call **IOleAdviseHolder::SendOnSave** whenever you save the object the advise holder is associated with.

To take the object from the running state to the loaded state, the client calls **IOleObject::Close**. Within that implementation, if the user wants to save the object to persistent storage, the object calls **IOleClientSite::SaveObject**, followed by the call to **IOleAdviseHolder::SendOnSave**.

See Also

IAdviseSink::OnSave

IOleAdviseHolder::Unadvise Quick Info

Deletes a previously established advisory connection.

```
HRESULT Unadvise(
  DWORD dwConnection      //Value identifying an established advisory connection
);
```

Parameter

dwConnection

[in] Contains a nonzero DWORD previously returned by **IOleAdviseHolder::Advise** in *pdwConnection*.

Return Values

S_OK

Advisory connection deleted successfully.

OLE_E_NOCONNECTION

The *dwConnection* parameter does not represent a valid advisory connection.

Remarks

IOleAdviseHolder::Unadvise is intended to be used to implement **IOleObject::Unadvise** to delete an advisory connection. In general, you would use the OLE advise holder having obtained a pointer through a call to **CreateOleAdviseHolder**.

Normally, containers call this method at shutdown or when an object is deleted. In certain cases, containers could call this method on objects that are running but not currently visible, as a way of reducing the overhead of maintaining multiple advisory connections.

See Also

IOleAdviseHolder::Advise, **IOleAdviseHolder::EnumAdvise**, **IOleObject::Unadvise**

IOleCache Quick Info

The **IOleCache** interface provides control of the presentation data that gets cached inside of an object. Cached presentation data is available to the container of the object even whenthe server application is not running or is unavailable.

When to Implement

The **IOleCache** interface can be implemented by an object handler and an in-process server. In general, however, the methods of **IOleCache** are implemented as part of the **IOleCache2** interface, which inherits the **IOleCache** definition, and adds methods for selectively updating the cache. Rather than implementing, however, it is typical to use or aggregate the OLE implementation, a pointer to which is available through a call to **CreateDataCache**.

When to Use

A client calls the methods of **IOleCache** to control what the data cache holds. A typical client would be an OLE Documents container that would cache an object's presentation so it is available without the object actually being active.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleCache Methods	Description
Cache	Adds a presentation to the data or view cache.
Uncache	Removes a presentation previously added with **IOleCache::Cache**.
EnumCache	Returns an object to enumerate the current cache connections.
InitCache	Fills the cache with all the presentation data from the data object.
SetData	Fills the cache with specified format of presentation data.

IOleCache::Cache Quick Info

Specifies the format and other data to be cached inside an embedded object.

```
HRESULT Cache(
  FORMATETC * pFormatetc,     //Pointer to the format and data to be cached
  DWORD advf,                 //Flags that control the caching
  DWORD * pdwConnection       //Pointer to the connection for future calls to uncache
);
```

Parameters

pFormatetc

[in] Pointer to the format and other data to be cached. View caching is specified by passing a zero clipboard format in *pFormatetc*.

advf

[in] Contains a group of flags that control the caching. Valid values can be derived by using an OR operation on the values in the **ADVF** enumeration. When used in this context, for a cache, these values have specific meanings, which are outlined in the Remarks. Refer to the **ADVF** enumeration for a more detailed description.

pdwConnection

[out] Pointer to a returned DWORD token that identifies this connection and can later be used to turn caching off (by passing it to **IOleCache::Uncache**). If this value is zero, the connection was not established. The OLE-provided implementation uses nonzero numbers for connection identifiers.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

Requested data or view successfully cached.

CACHE_S_FORMATETC_NOTSUPPORTED

The cache was created, but the object application does not support the specified format. Cache creation succeeds even if the format is not supported, allowing the caller to fill the cache. If, however, the caller does not need to keep the cache, call **IOleCache::UnCache**.

CACHE_S_SAMECACHE

A cache already exists for the **FORMATETC** passed to **IOleCache::Uncache**. In this case, the new advise flags are assigned to the cache, and the previously assigned connection identifier is returned.

DV_E_LINDEX

Invalid value for *lindex*; currently only -1 is supported.

DV_E_TYMED

The value is not valid for *pFormatetc->tymed*.

DV_E_DVASPECT

The value is not valid for *pFormatetc->dwAspect*.

DV_E_CLIPFORMAT

The value is not valid for *pFormatetc->cfFormat*.

CO_E_NOTINITIALIZED

The cache's storage is not initialized.

DV_E_DVTARGETDEVICE

The value is not valid for *pFormatetc->ptd*.

OLE_E_STATIC

The cache is for a static object and it already has a cache node.

Remarks

IOleCache::Cache can specify either data caching or view (presentation) caching. To specify data caching, a valid data format must be passed in *pFormatetc*. For view caching, the cache object itself decides on the format to cache, so a caller would pass a zero data format in *pFormatetc* as follows:

```
pFormatetc->cfFormat == 0
```

A custom object handler can choose not to store data in a given format. Instead, it can synthesize it on demand when requested.

The *advf* value specifies a member of the ADVF enumeration. When one of these values (or an OR'd combination of more than one value) is used in this context, these values mean the following:

ADVF Value	Description
ADVF_NODATA	The cache is not to be updated by changes made to the running object. Instead, the container will update the cache by explicitly calling **IOleCache::SetData**, **IDataObject::SetData**, or **IOleCache2::UpdateCache**. This flag is usually used when the iconic aspect of an object is being cached.
ADVF_ONLYONCE	Update the cache one time only. After the update is complete, the advisory connection between the object and the cache is disconnected. The source object for the advisory connection calls the **IAdviseSink::Release** method.
ADVF_PRIMEFIRST	The object is not to wait for the data or view to change before updating the cache. OR'd with ADVF_ONLYONCE, this parameter provides an asynchronous **GetData** call.
ADVFCACHE_NOHANDLER	Synonym for

	ADVFCACHE_FORCEBUILTIN.
ADVFCACHE_FORCEBUILTIN	Used by DLL object applications and object handlers that draw their objects to cache presentation data to ensure that there is a presentation in the cache. This ensures that the data can be retrieved even when the object or handler code is not available.
ADVFCACHE_ONSAVE	Updates the cached representation only when the object containing the cache is saved. The cache is also updated when the OLE object changes from the running state back to the loaded state (because a subsequent save operation would require running the object again).

See Also

ADVF, **IOleCache::Uncache**

IOleCache::EnumCache Quick Info

Returns a pointer to an enumeration object that can be used to enumerate the current cache connections.

```
HRESULT EnumCache(

  IEnumSTATDATA ** ppenumSTATDATA       //Indirect pointer to enumerator object
);
```

Parameter

ppenumSTATDATA

[out] Indirect pointer to the **IEnumSTATDATA** interface on the new enumerator object. If this value is NULL, there are currently no cache connections at this time.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The enumerator object is successfully instantiated or there are no cache connections. Check the *ppenumSTATDATA* parameter to determine which result occurred. If the *ppenumSTATDATA* parameter is NULL, then there are no cache connections at this time.

Remarks

The enumerator object returned by this method implements the **IEnumSTATDATA** interface, one of the standard enumerator interfaces that contain the **Next**, **Reset**, **Clone**, and **Skip** methods. **IEnumSTATDATA** enumerates the data stored in an array of **STATDATA** structures containing information about current cache connections.

See Also

IEnum*XXXX*, **IEnumSTATDATA**, **IOleCache::Cache**

IOleCache::InitCache Quick Info

Fills the cache as needed using the data provided by the specified data object.

```
HRESULT InitCache(
  IDataObject * pDataObject      //Pointer to the data object from which the cache is initialized
);
```

Parameter

pDataObject

[in] Pointer to the **IDataObject** interface on the data object from which the cache is to be initialized.

Return Values

This method supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The cache was filled using the data provided.

OLE_E_NOTRUNNING

The cache is not running.

CACHE_E_NOCACHE_UPDATED

None of the caches were updated.

CACHE_S_SOMECACHES_NOTUPDATED

Only some of the existing caches were updated.

Remarks

IOleCache::InitCache is usually used when creating an object from a drag-and-drop operation or from a clipboard paste operation. It fills the cache as needed with presentation data from all the data formats provided by the data object provided on the clipboard or in the drag-and-drop operation. Helper functions like **OleCreateFromData** or **OleCreateLinkFromData** call this method when needed. If a container does not use these helper functions to create compound document objects, it can use **IOleCache::Cache** to set up the cache entries which are then filled by **IOleCache::InitCache**.

See Also

IOleCache::Cache

IOleCache::SetData Quick Info

Initializes the cache with data in a specified format and on a specified medium.

```
HRESULT SetData(
  FORMATETC * pFormatetc,     //Pointer to the format of the presentation data to be placed in the cache
  STGMEDIUM * pmedium,        //Pointer to the storage medium containing the data to be placed in the cache
  BOOL fRelease               //Ownership of the storage medium after this method is completed
);
```

Parameters

pFormatetc

[in] Pointer to the format of the presentation data being placed in the cache.

pmedium

[in] Pointer to the storage medium that contains the presentation data.

fRelease

[in] Ownership of the storage medium after completion of the method. If *fRelease* is TRUE, the cache takes ownership, freeing the medium when it is finished using it. When *fRelease* is FALSE, the caller retains ownership and is responsible for freeing the medium. The cache can only use the storage medium for the duration of the call.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The cache was filled.

DV_E_LINDEX

The value is not valid for *pFormatetc->lindex.* Currently, only -1 is supported.

DV_E_FORMATETC

The **FORMATETC** structure is invalid.

DV_E_TYMED

The value is not valid for *pFormatetc->tymed*.

DV_E_DVASPECT

The value is not valid for *pFormatetc->dwAspect*.

OLE_E_BLANK

Uninitialized object.

DV_E_TARGETDEVICE

The object is static and *pFormatetc->ptd* is non-NULL.

STG_E_MEDIUMFULL

The storage medium is full.

Remarks

IOleCache::SetData is usually called when an object is created from the Clipboard or through a drag-and-drop operation, and Embed Source data is used to create the object.

IOleCache::SetData and **IOleCache::InitCache** are very similar. There are two main differences. The first difference is that while **IOleCache::InitCache** initializes the cache with the presentation format provided by the data object, **IOleCache::SetData** initializes it with a single format. Second, the **IOleCache::SetData** method ignores the ADVF_NODATA flag while **IOleCache::InitCache** obeys this flag.

A container can use this method to maintain a single aspect of an object, such as the icon aspect of the object.

See Also

IDataObject::SetData, **IOleCache::Cache**, ADVF enumeration

IOleCache::Uncache Quick Info

Removes a cache connection created in a prior call to **IOleCache::Cache**.

```
HRESULT Uncache(
  DWORD dwConnection      //Cache connection to remove
);
```

Parameter

dwConnection

[in] Cache connection to remove. This nonzero value was returned by **IOleCache::Cache** when the cache was originally established.

Return Values

S_OK

The cache connection was deleted.

OLE_E_NOCONNECTION

No cache connection exists for *dwConnection*.

Remarks

The **IOleCache::Uncache** method removes a cache connection that was created in a prior call to **IOleCache::Cache**. It uses the *dwConnection* parameter that was returned by the prior call to **IOleCache::Cache**.

See Also

IOleCache::Cache

IOleCache2 Quick Info

The **IOleCache2** interface allows object clients to selectively update each cache that was created with **IOleCache::Cache**.

When to Implement

The **IOleCache2** interface can be implemented by an object handler and an in-process server. Rather than implementing, however, it is typical to use or aggregate the OLE implementation in the default handler, a pointer to which is available through a call to **CreateDataCache**.

When to Use

The OLE-provided implementation includes implementations of the **IOleCache** interface methods, from which **IOleCache2** inherits its contract definition. The **IOleCache2** interface is called by container applications, object handlers, or in process servers to update one or more of the caches that were created with the **IOleCache::Cache** method. This extended interface was added so client applications can exercise precise control over updates to the caches being maintained.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleCache Methods	Description
Cache	Adds a presentation to the data or view cache.
Uncache	Removes a presentation previously added with **IOleCache::Cache**.
EnumCache	Returns an object to enumerate the current cache connections.
InitCache	Fills the cache with all the presentation data from the data object.
SetData	Fills the cache with specified format of presentation data.

IOleCache2 Methods	Description
UpdateCache	Updates the specified cache(s).
DiscardCache	Discards cache(s) found in memory.

IOleCache2::DiscardCache Quick Info

Discards the caches in memory.

```
HRESULT DiscardCache(
  DWORD dwDiscardOptions      //Save options
);
```

Parameter

dwDiscardOptions

[in] DWORD value from the **DISCARDCACHE** enumeration that indicates whether data is to be saved prior to being discarded. Containers that have drawn a large object and need to free up memory can specify DISCARDCACHE_SAVEIFDIRTY so that the newest presentation is saved for the next time the object must be drawn.

Containers that have activated an embedded object, made some changes, and then called **IOleObject::Close(**OLECLOSE_NOSAVE) to roll back the changes can specify DISCARDCACHE_NOSAVE to ensure that the native and presentation data are not out of synchronization.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The cache(s) were discarded according to the value specified in *dwDiscardOptions*.

OLE_E_NOSTORAGE

There is no storage available for saving the data in the cache.

STG_E_MEDIUMFULL

The storage medium is full.

Remarks

The **IOleCache2::DiscardCache** method is commonly used to handle low memory conditions by freeing memory currently being used by presentation caches.

Once discarded, the cache will satisfy subsequent **IDataObject::GetData** calls by reverting to disk-based data.

See Also

DISCARDCACHE, **IOleCache**, **IOleCacheControl**

IOleCache2::UpdateCache Quick Info

Updates specified cache(s). It updates the cache according to the value of a parameter. This method is used when the application needs precise control over caching.

```
HRESULT UpdateCache(
  IDataObject * pDataObject,    //Pointer to the data object from which the cache is updated
  DWORD grfUpdt,                //Type of cache to update
  LPVOID pReserved              //Reserved
);
```

Parameters

pDataObject

[in] Pointer to the **IDataObject** interface on the data object from which the cache is updated. Object handlers and in-process servers typically pass a non-NULL value. A container application usually passes NULL, and the source is obtained from the currently running object.

grfUpdt

[in] Type of cache to update. The value is obtained by combining values from the following table:

Cache Control Values	Description
UPDFCACHE_NODATACACHE	Updates caches created by using ADVF_NODATA in the call to **IOleCache::Cache**.
UPDFCACHE_ONSAVECACHE	Updates caches created by using ADVFCACHE_ONSAVE in the call to **IOleCache::Cache**.
UPDFCACHE_ONSTOPCACHE	Updates caches created by using ADVFCACHE_ONSTOP in the call to **IOleCache::Cache**.
UPDFCACHE_NORMALCACHE	Dynamically updates the caches (as is normally done when the object sends out OnDataChange notices).
UPDFCACHE_IFBLANK	Updates the cache if blank, regardless of any other flag specified.
UPDFCACHE_ONLYIFBLANK	Updates only caches that are blank.
UPDFCACHE_IFBLANKORONSAVECACHE	The equivalent of using an OR operation to combine UPDFCACHE_IFBLANK and UPDFCACHE_ONSAVECACHE.
UPDFCACHE_ALL	Updates all caches.
UPDFCACHE_ALLBUTNODATACACHE	Updates all caches except those created with ADVF_NODATA in the call to **IOleCache::Cache**. Thus, you can control updates to the caches created with the ADVF_NODATA flag and only

update these caches explicitly.

pReserved

[in] Reserved for future use; must be NULL.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The cache(s) were updated according to the value specified in *grfUpdt*.

OLE_E_NOTRUNNING

The specified *pDataObject* is not running.

CACHE_E_NOCACHE_UPDATED

None of the caches were updated.

CACHE_S_SOMECACHES_NOTUPDATED

Some of the caches were updated.

See Also

IOleCache, **IOleCacheControl**, **IDataObject**

IOleCacheControl Quick Info

The **IOleCacheControl** interface provides proper maintenance of caches. It maintains the caches by connecting the running object's **IDataObject** implementation to the cache, allowing the cache to receive notifications from the running object.

When to Implement

The OLE-provided implementation is used by most handlers and in-process servers. You can get a pointer to the OLE data cache object through a call to **CreateDataCache**.

When to Use

Object handlers and in-process servers use this interface internally to connect the cache part of the handler to the **IDataObject** implementation on the running object. Container applications have no need for this interface; they use **IRunnableObject** or **OleRun** instead.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleCacheControl Methods	Description
OnRun	Notifies the cache when the data object is running so the cache object can establish advise sinks as needed.
OnStop	Notifies the cache to terminate any existing advise sinks.

IOleCacheControl::OnRun Quick Info

Notifies the cache that the data source object has now entered its running state.

```
HRESULT OnRun(
  DATAOBJECT * pDataObject      //Pointer to the object that is now running
);
```

Parameter

pDataObject

[in] Pointer to the **IDataObject** interface on the object that is entering the running state.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The cache was notified and *pDataObject* is valid.

Remarks

When **IOleCacheControl::OnRun** is called, the cache sets up advisory connections as necessary with the source data object so it can receive notifications. The advisory connection created between the running object and the cache is destroyed when **IOleCacheControl::OnStop** is called.

Some object handlers or in-process servers might use the cache passively, and not call **IOleCacheControl::OnRun**. These applications must call **IOleCache2::UpdateCache**, **IOleCache::InitCache**, or **IOleCache::SetData** to fill the cache when necessary to ensure that the cache gets updated.

IOleCacheControl::OnRun does not add a reference count on the pointer to **IDataObject** passed in *pDataObject.* Because it is the responsibility of the caller of **OleRun** to ensure that the lifetime of the *pDataObject* pointer lasts until **OnStop** is called, the caller must be holding a pointer to **IDataObject** on the data object of interest.

See Also

IOleCache2::UpdateCache, **IOleCacheControl::OnStop**

IOleCacheControl::OnStop Quick Info

Notifies the cache it should terminate any existing connection previously given to it by using **IOleCacheControl::OnRun**. No indication is given as to whether a connection existed or not.

HRESULT OnStop( );

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The cache was notified and the advisory connection was successfully removed.

Remarks

The data advisory connection between the running object and the cache is destroyed as part of calling **IOleCacheControl::OnStop**.

See Also

IOleCacheControl::OnRun

IOleClientSite Quick Info

The **IOleClientSite** interface is the primary means by which an embedded object obtains information about the location and extent of its display site, its moniker, its user interface, and other resources provided by its container. An object server calls **IOleClientSite** to request services from the container. A container must provide one instance of **IOleClientSite** for every compound-document object it contains.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IOleClientSite Methods	Description
SaveObject	Saves embedded object.
GetMoniker	Requests object's moniker.
GetContainer	Requests pointer to object's container.
ShowObject	Asks container to display object.
OnShowWindow	Notifies container when object becomes visible or invisible
RequestNewObjectLayout	Asks container to resize display site.

IOleClientSite::GetContainer Quick Info

Returns a pointer to the container's **IOleContainer** interface.

```
HRESULT GetContainer(
  LPOLECONTAINER FAR* ppContainer      //Indirect pointer to the interface on the object
);
```

Parameter

ppContainer

[out] Indirect pointer to where the **IOleContainer** interface on the object should be returned. If an error is returned, this parameter must be set to NULL.

Return Values

S_OK

The pointer to the container's **IOleContainer** interface was successfully returned.

OLE_E_NOT_SUPPORTED

Client site is in OLE 1 container.

E_NOINTERFACE

The container does not implement the **IOleContainer** interface.

Remarks

If a container supports links to its embedded objects, implementing **IOleClientSite::GetContainer** enables link clients to enumerate the container's objects and recursively traverse a containment hierarchy. This method is optional but recommended for all containers that expect to support links to their embedded objects.

Link clients can traverse a hierarchy of compound-document objects by recursively calling **IOleClientSite::GetContainer** to get a pointer to the link source's container; followed by **IOleContainer::QueryInterface** to get a pointer to the container's **IOleObject** interface and, finally, **IOleObject::GetClientSite** to get the container's client site in its container.

Simple containers that do not support links to their embedded objects probably do not need to implement this method. Instead, they can return E_NOINTERFACE and set *ppContainer* to NULL.

IOleClientSite::GetMoniker Quick Info

Returns a moniker to an object's client site. An object can force the assignment of its own or its container's moniker by specifying a value for *dwAssign*.

```
HRESULT GetMoniker(
  DWORD dwAssign,          //Value specifying how moniker is assigned
  DWORD dwWhichMoniker,    //Value specifying which moniker is assigned
  IMoniker ** ppmk         //Indirect pointer to moniker
);
```

Parameters

dwAssign

[in] Specifies whether to get a moniker only if one already exists, force assignment of a moniker, create a temporary moniker, or remove a moniker that has been assigned. In practice, you will usually request that the container force assignment of the moniker. Values defining these choices are contained in the enumeration **OLEGETMONIKER**.

dwWhichMoniker

[in] DWORD that specifies whether to return the container's moniker, the object's moniker relative to the container, or the object's full moniker. In practice, you will usually request the object's full moniker. Values defining these choices are contained in the enumeration **OLEWHICHMK**.

ppmk

[out] Indirect pointer to the **IMoniker** interface on the moniker for the object's client site. If an error is returned, this parameter must be set to NULL. Each time a container receives a call to **IOleClientSite::GetMoniker**, it must increase the reference count on the pointer it returns. It is the caller's responsibility to call **Release** when it is done with the pointer.

Return Values

This method supports the standard return values E_FAIL and E_UNEXPECTED, as well as the following:

S_OK

Requested moniker returned successfully.

E_NOTIMPL

This container cannot assign monikers to objects. This is the case with OLE 1 containers.

Remarks

Containers implement **IOleClientSite::GetMoniker** as a way of passing out monikers for their embedded objects to clients wishing to link to those objects.

When a link is made to an embedded object or to a pseudo-object within it (a range of cells in a spreadsheet, for example), the object needs a moniker to construct the composite moniker indicating the source of the link. If the embedded object does not already have a moniker, it can call **IOleClientSite::GetMoniker** to request one.

Every container that expects to contain links to embeddings should support **IOleClientSite::GetMoniker**

to give out OLEWHICHMK_CONTAINER, thus enabling link tracking when the link client and link source files move, but maintain the same relative position.

An object must not persistently store its full moniker or its container's moniker, because these can change while the object is not loaded. For example, either the container or the object could be renamed, in which event, storing the container's moniker or the object's full moniker would make it impossible for a client to track a link to the object.

In some very specialized cases, an object may no longer need a moniker previously assigned to it and may wish to have it removed as an optimization. In such cases, the object can call **IOleClientSite::GetMoniker** with OLEGETMONIKER_UNASSIGN to have the moniker removed.

See Also

IOleObject::GetMoniker, **IOleObject::SetMoniker**

IOleClientSite::OnShowWindow Quick Info

Notifies a container when an embedded object's window is about to become visible or invisible. This method does not apply to an object that is activated in place and therefore has no window separate from that of its container.

```
HRESULT OnShowWindow(
  BOOL fShow      //Value indicating if window is becoming visible
);
```

Parameter

fShow

[in] Value that indicates whether an object's window is open (TRUE) or closed (FALSE).

Return Value

S_OK

Shading or hatching has been successfully added or removed.

Remarks

An embedded object calls **IOleClientSite::OnShowWindow** to keep its container informed when the object is open in a window. This window may or may not be currently visible to the end user. The container uses this information to shade the object's client site when the object is displayed in a window, and to remove the shading when the object is not. A shaded object, having received this notification, knows that it already has an open window and therefore can respond to being double-clicked by bringing this window quickly to the top, instead of launching its application in order to obtain a new one.

IOleClientSite::RequestNewObjectLayout Quick Info

Asks container to allocate more or less space for displaying an embedded object.

HRESULT RequestNewObjectLayout();

Return Values

S_OK

Request for new layout succeeded.

E_NOTIMPL

Client site does not support requests for new layout.

Remarks

Currently, there is no standard mechanism by which a container can negotiate *how much* room an object would like. When such a negotiation is defined, responding to this method will be optional for containers.

IOleClientSite::SaveObject Quick Info

Saves the object associated with the client site. This function is synchronous; by the time it returns, the save will be completed.

HRESULT SaveObject();

Parameter

HRESULT **SaveObject(***void***)**

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The object was saved.

Remarks

An embedded object calls **IOleClientSite::SaveObject** to ask its container to save it to persistent storage when an end user chooses the File Update or Exit commands. The call is synchronous, meaning that by the time it returns, the save operation will be completed.

Calls to **IOleClientSite::SaveObject** occur in most implementations of **IOleObject::Close**. Normally, when a container tells an object to close, the container passes a flag specifying whether the object should save itself before closing, prompt the user for instructions, or close without saving itself. If an object is instructed to save itself, either by its container or an end user, it calls **IOleClientSite::SaveObject** to ask the container application to save the object's contents before the object closes itself. If a container instructs an object not to save itself, the object should not call **SaveObject**.

See Also

IOleObject::Close

IOleClientSite::ShowObject Quick Info

Tells the container to position the object so it is visible to the user. This method ensures that the container itself is visible and not minimized.

HRESULT ShowObject();

Return Values

S_OK

Container has tried to make the object visible.

OLE_E_NOT_SUPPORTED

Client site is in an OLE 1 container.

Remarks

After a link client binds to a link source, it commonly calls **IOleObject::DoVerb** on the link source, usually requesting the source to perform some action requiring that it display itself to the user. As part of its implementation of **DoVerb**, the link source can call **IOleClientSite::ShowObject**, which forces the client to show the link source as best it can. If the link source's container is itself an embedded object, it will recursively invoke **IOleClientSite::ShowObject** on its own container.

Having called the **ShowObject** method, a link source has no guarantee of being appropriately displayed because its container may not be able to do so at the time of the call. The **ShowObject** method does not guarantee visibility, only that the container will do the best it can.

See Also

IOleObject::DoVerb

IOleContainer Quick Info

The **IOleContainer** interface is used to enumerate objects in a compound document or lock a container in the running state. Container and object applications both implement this interface.

When to Implement

Applications that support links and links to embedded objects implement this interface to provide object enumeration, name parsing, and silent updates of link sources. Simple, nonlinking containers do not need to implement **IOleContainer** if it is useful mainly to support links to embedded objects.

When to Use

Call **IOleContainer** to enumerate the objects in a compound document or to lock a container so that silent updates of link sources can be carried out safely.

Many applications inherit the functions of **IOleContainer** by implementing **IOleItemContainer**, which is used to bind item monikers.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IParseDisplayName Method	Description
ParseDisplayName	Parses object's display name to form moniker.

IOleContainer Methods	Description
EnumObjects	Enumerates objects in a container.
LockContainer	Keeps container running until explicitly released.

See Also

IOleItemContainer, IParseDisplayName

IOleContainer::EnumObjects Quick Info

Enumerates objects in the current container.

```
HRESULT EnumObjects(
    DWORD grfFlags,              //Value specifying what is to be enumerated
    IEnumUnknown **ppenum        //Indirect pointer to enumerator object
);
```

Parameters

grfFlags

[in] Value that specifies which objects in a container are to be enumerated, as defined in the enumeration **OLECONTF**.

ppenum

[out] When successful, indirect pointer to the **IEnumUnknown** interface on the enumerator object. Each time a container receives a successful call to **EnumObjects**, it must increase the reference count on the pointer the method returns. It is the caller's responsibility to call **IUnknown::Release** when it is done with the pointer. If an error is returned, this parameter must be set to NULL.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

Enumerator successfully returned.

E_NOTIMPL

Object enumeration not supported.

Remarks

A container should implement **EnumObjects** to enable programmatic clients to find out what objects it holds. This method, however, is not called in standard linking scenarios.

See Also

IEnumUnknown, **IOleItemContainer**, **OLECONTF**

IOleContainer::LockContainer Quick Info

Keeps an embedded object's container running.

```
HRESULT LockContainer(
  BOOL fLock      //Value indicating lock or unlock
);
```

Parameter

fLock

[in] Value that specifies whether to lock (TRUE) or unlock (FALSE) a container.

Return Values

This method supports the standard return values E_FAIL and E_OUTOFMEMORY, as well as the following:

S_OK

Container was locked successfully.

Remarks

An embedded object calls **IOleContainer::LockContainer** to keep its container running when the object has link clients that require an update. If an end user selects File Close from the container's menu, however, the container ignores all outstanding **LockContainer** locks and closes the document anyway.

Notes to Callers

When an embedded object changes from the loaded to the running state, it should call **IOleContainer::LockContainer** with the *fLock* parameter set to TRUE. When the embedded object shuts down (transitions from running to loaded), it should call **IOleContainer::LockContainer** with the *fLock* parameter set to FALSE.

Each call to **LockContainer** with *fLock* set to TRUE must be balanced by a call to **LockContainer** with *fLock* set to FALSE. Object applications typically need not call **LockContainer**; the default handler makes these calls automatically for object applications implemented as .EXEs as the object makes the transition to and from the running state. Object applications not using the default handler, such as DLL object applications, must make the calls directly.

An object should have no strong locks on it when it registers in the Running Object Table, but it should be locked as soon as the first external client connects to it. Therefore, following registration of the object in the Running Object Table, object handlers and DLL object applications, as part of their implementation of **IRunnableObject::Run**, should call **IOleContainer::LockContainer(**TRUE**)** to lock the object.

Notes to Implementers

The container must keep track of whether and how many calls to **LockContainer(**TRUE**)** have been made. To increment or decrement the reference count, **IOleContainer::LockContainer** calls **CoLockObjectExternal** with a flag set to match *fLock*.

See Also

CoLockObjectExternal, **IRunnableObject::Run**

IOleControl Quick Info

The **IOleControl** interface provides the features for supporting keyboard mnemonics (**GetControlInfo**, **OnMnemonic**), ambient properties (**OnAmbientPropertyChange**), and events (**FreezeEvents**) in control objects.

When to Implement

Implement this interface for a control object to communicate with the control's container, for example, when managing keyboard activity or obtaining the container's ambient properties.

When to Use

A control container uses this interface to work with keyboard mnemonics, ambient properties, and events of a contained control object.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IOleControl Methods	Description
GetControlInfo	Fills in a **CONTROLINFO** structure with information about the control's keyboard behavior.
OnMnemonic	Informs the control that the user has pressed a keystroke that the control specified through **GetControlInfo**. The control takes whatever action is appropriate for the keystroke.
OnAmbientPropertyChange	Informs an control that one or more of the container's ambient properties has changed.
FreezeEvents	Indicates whether or not the container ignores or accepts events from the control.

See Also

IOleControlSite, ISimpleFrameSite

IOleControl::FreezeEvents Quick Info

Indicates whether the container is ignoring or accepting events from the control.

```
HRESULT FreezeEvents(
    BOOL bFreeze      //Indicates whether to ignore or process events
);
```

Parameters

bFreeze

[out] Indicates whether the container will ignore (TRUE) or now process (FALSE) events from the control.

Return Values

S_OK

Returned in all cases.

Remarks

The control is not required to stop sending events when *bFreeze* is TRUE. However, the container is not going to process them in this case. If a control depends on the container's processing -- as with request events that return information from the container -- the control must either discard the event or queue the event to send later when **IOleControl::FreezeEvents** returns FALSE.

Notes to Implementers

As with **IOleControl::OnAmbientPropertyChange**, S_OK is returned in all cases in order to prevent a container from making assumptions about a control's behavior based on return values.

IOleControl::GetControlINfo

Fills in a **CONTROLINFO** structure with information about a control's keyboard mnemonics and keyboard behavior.

```
HRESULT GetControlInfo(
  CONTROLINFO* pCI      //Pointer to structure
);
```

Parameters

pCI

[in, out] Pointer to the caller-allocated **CONTROLINFO** structure to be filled in.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The structure was filled successfully.

E_NOTIMPL

The control has no mnemonics.

E_POINTER

The address in *pCI* is not valid. For example, it may be NULL.

See Also

CONTROLINFO

IOleControl::OnAmbientPropertyChange Quick Info

Informs a control that one or more of the container's ambient properties (available through the control site's **IDispatch**) has changed.

```
HRESULT OnAmbientPropertyChange(
  DISPID dispID      //Dispatch identifier of the ambient property
);
```

Parameters

dispID

[in] Dispatch identifier of the ambient property that changed. If the *dispID* parameter is DISPID_UNKNOWN, it indicates that multiple properties changed. In this case, the control should check all the ambient properties of interest to obtain their current values.

Return Values

S_OK

Returned in all cases.

Remarks

Notes to Implementers

S_OK is returned in all cases even when the control does not support ambient properties or some other failure has occurred. The caller sending the notification cannot attempt to use an error code (such as E_NOTIMPL) to determine whether to send the notification in the future. Such semantics are not part of this interface.

IOleControl::OnMnemonic Quick Info

Informs a control that the user has pressed a keystroke that matches one of the **ACCEL** entries in the mnemonic table returned through **IOleControl::GetControlInfo**. The control takes whatever action is appropriate for the keystroke.

```
HRESULT OnMnemonic(
  LPMSG pMsg      //Pointer to structure
);
```

Parameters

pMsg

[in] Pointer to the **MSG** structure describing the keystroke to be processed.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The control accepted the mnemonic.

E_NOTIMPL

The control doesn't handle any mnemonics. This indicates an unexpected condition and a caller error. For example, the caller has mismatched which control has which mnemonic.

Remarks

Notes to Implementers

If a control changes the contents of its **CONTROLINFO** structure, it must notify its container by calling **IOleControlSite::OnControlInfoChanged**.

Notes to Callers

A container of a control is allowed to cache the control's **CONTROLINFO** structure, provided that the container implements **IOleControlSite::OnControlInfoChanged** to know when it must update its cached information.

See Also

IOleControl::GetControlInfo, **IOleControlSite::OnControlInfoChanged**

IOleControlSite Quick Info

The **IOleControlSite** interface provides the methods that enable a site object to manage each embedded control within a container. A site object provides **IOleControlSite** as well as other site interfaces such as **IOleClientSite** and **IOleInPlaceSite**. When a control requires the services expressed through this interface, it will query one of the other client site interfaces for **IOleControlSite**.

When to Implement

Implement this interface on an in-place capable site object to support the embedding of controls in the site.

When to Use

A control uses this interface to work with a control-aware container.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IOleControlSite Methods	Description
OnControlInfoChanged	Informs the container that the control's **CONTROLINFO** structure has changed and that the container should call the control's **IOleControl::GetControlInfo** for an update.
LockInPlaceActive	Indicates whether or not this control should remain in-place active, regardless of possible deactivation events.
GetExtendedControl	Requests an **IDispatch** pointer to the extended control that the container uses to wrap the real control.
TransformCoords	Converts between a **POINTL** structure expressed in HIMETRIC units (as is standard in OLE) and a **POINTF** structure expressed in units the container specifies.
TranslateAccelerator	Instructs the container to process a specified keystroke.
OnFocus	Indicates whether the embedded control in this control site has gained or lost the focus.
ShowPropertyFrame	Instructs the container to show a property page frame for the control object if the container so desires.

See Also

IOleControl, ISimpleFrameSite

IOleControlSite::GetExtendedControl Quick Info

Requests an **IDispatch** pointer to the extended control that the container uses to wrap the real control.

```
HRESULT GetExtendedControl(
  IDispatch** ppDisp      //Indirect pointer to the interface of the extended control
);
```

Parameters

ppDisp

[out] Indirectly pointer to the extended control's **IDispatch** interface. This parameter is set to NULL on failure. On success, the caller is responsible for calling **IDispatch::Release** when this pointer is no longer needed.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The extended control's **IDispatch** is returned in *ppDisp*.

E_NOTIMPL

The container does not implement extended controls.

E_POINTER

The address in *ppDisp* or **ppDisp* is not valid. For example, it may be NULL.

Remarks

This method gives the real control access to whatever properties and methods the container maintains in the extended control. These properties and methods would otherwise be inaccessible to the control.

Notes to Callers

The returned pointer is the responsibility of the caller, which must release it when it is no longer needed.

IOleControlSite::LockInPlaceActive Quick Info

Indicates whether or not a control should remain in-place active. Calls to this method typically nest an event to ensure that the object's activation state remains stable throughout the processing of the event.

```
HRESULT LockInPlaceActive(
  BOOL fLock      //Indicates whether to ensure the active state
);
```

Parameters

fLock

[in] Indicates whether to ensure the in-place active state (TRUE) or to allow activation to change (FALSE). When TRUE, a supporting container must not deactivate the in-place object until this method is called again with FALSE.

Return Values

S_OK

The lock or unlock was made successfully.

E_NOTIMPL

The container does not support in-place locking.

Remarks

This method affects the control's in-place active state but not its UI-active state.

IOleControlSite::OnControlInfoChanged Quick Info

Informs the container that the control's **CONTROLINFO** structure has changed and that the container should call the control's **IOleControl::GetControlInfo** for an update.

HRESULT OnControlInfoChanged(void);

Return Values

S_OK

Returned in all cases.

See Also

IOleControl::GetControlInfo

IOleControlSite::OnFocus Quick Info

Indicates whether the control managed by this control site has gained or lost the focus, according to the *fGotFocus* parameter. The container uses this information to update the state of Default and Cancel buttons according to how the control with the focus processes Return or Esc keys. A control's behavior regarding the Return and Esc keys is specified in the control's **CONTROLINFO** structure. See **IOleControl::GetControlInfo**.

```
HRESULT OnFocus(

  BOOL fGotFocus      //Indicates whether the control gained focus
);
```

Parameters

fGotFocus

[in] Indicates whether the control gained (TRUE) or lost the focus (FALSE).

Return Values

S_OK

Returned in all cases.

See Also

IOleControl::GetControlInfo

IOleControlSite::ShowPropertyFrame Quick Info

Instructs a container to display a property sheet for the control embedded in this site.

HRESULT ShowPropertyFrame(void);

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The container successfully displayed property pages for this control and the control must not show its own.

E_NOTIMPL

The container does not need to show property pages itself.

Remarks

A control must always call this method in the container first when it intends to show its own property pages. Calling this method gives the container a chance to have those property pages work with the container's extended controls. The container may include its own property pages as well in such cases, which doesn't affect the control at all. If the container does not implement this method or if it returns a failure of any kind, the control can show its property pages directly. Otherwise, the container has shown the pages.

IOleControlSite::TransformCoords Quick Info

Converts between a **POINTL** structure expressed in HIMETRIC units (as is standard in OLE) and a **POINTF** structure expressed in units specified by the container. By converting the methods, the control can ensure that it sends coordinate information to the container in units that are directly usable in the container without additional conversion.

```
HRESULT TransformCoords(
  POINTL* pptlHimetric ,      //Indirect pointer to structure
  POINTF* pptfContainer ,     //Indirect pointer to structure
  DWORD dwFlags               //Flags indicating the exact conversion
);
```

Parameters

pptlHimetric

[in, out] Indirect pointer to a **POINTL** structure containing coordinates expressed in HIMETRIC units. This is an [in] parameter when *dwFlags* contains XFORMCOORDS_HIMETRICTOCONTAINER; it is [out] with XFORMCOORDS_CONTAINERTOHIMETRIC. In the latter case, the contents are undefined on error.

pptfContainer

[in, out] Indirect pointer to a caller-allocated **POINTF** structure that receives the converted coordinates. This is an [in] parameter when *dwFlags* contains XFORMCOORDS_CONTAINERTOHIMETRIC; it is [out] with XFORMCOORDS_HIMETRICTOCONTAINER. In the latter case, the contents are undefined on error.

dwFlags

[in] Flags indicating the exact conversion to perform. The *dwFlags* parameter can be any combination of the following values except where indicated:

Flag Value	Description
XFORMCOORDS_POSITION	The coordinates to convert represent a position point. Cannot be used with XFORMCOORDS_SIZE.
XFORMCOORDS_SIZE	The coordinates to convert represent a set of dimensions. Cannot be used with XFORMCOORDS_POSITION.
XFORMCOORDS_HIMETRICTOCONTAINER	The operation converts *pptlHimetric* into *pptfContainer*. Cannot be used with XFORMCOORDS_CONTAINERTOHIMETRIC.
XFORMCOORDS_CONTAINERTOHIMETRIC	The operation converts *pptfContainer* into *pptlHimetric*. Cannot be used with XFORMCOORDS_HIMETRICTOCONTAINER.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The conversion was performed successfully.

E_NOTIMPL

The container does not require any special coordinate conversions. The container deals completely in HIMETRIC.

E_POINTER

The address in *pptlHimetric* or *pptfContainer* is not valid. For example, it may be NULL.

Remarks

A control uses this method when it has to send coordinates to a container within an event or some other custom call or when the control has container coordinates that it needs to convert into HIMETRIC units.

See Also

POINTF

IOleControlSite::TranslateAccelerator Quick Info

Instructs the control site to process the keystroke described in *pMsg* and modified by the flags in *grfModifiers*.

```
HRESULT TranslateAccelerator(
  LPMSG pMsg ,          //Pointer to the structure
  DWORD grfModifiers    //Flags describing the state of the keys
);
```

Parameters

pMsg

[in] Pointer to the **MSG** structure describing the keystroke to be processed.

grfModifiers

[in] Flags describing the state of the Control, Alt, and Shift keys. The value of the flag can be any valid **KEYMODIFIERS** enumeration values.

Return Values

S_OK

The container processed the message.

S_FALSE

The container did not process the message. This value must also be returned in all other error cases besides E_NOTIMPL.

E_NOTIMPL

The container does not implement accelerator support.

Remarks

This method is called by a control that can be UI-active. In such cases, a control can process all keystrokes first through **IOleInPlaceActiveObject::TranslateAccelerator**, according to normal OLE Compound Document rules. Inside that method, the control can give the container certain messages to process first by calling **IOleControlSite::TranslateAccelerator** and using the return value to determine if any processing took place. Otherwise, the control always processes the message first. If the control does not use the keystroke as an accelerator, it passes the keystroke to the container through this method.

See Also

IOleInPlaceActiveObject::TranslateAccelerator, **KEYMODIFIERS**

IOleInPlaceActiveObject Quick Info

The **IOleInPlaceActiveObject** interface provides a direct channel of communication between an in-place object and the associated application's outer-most frame window and the document window within the application that contains the embedded object. The communication involves the translation of messages, the state of the frame window (activated or deactivated), and the state of the document window (activated or deactivated). Also, it informs the object when it needs to resize its borders, and manages modeless dialog boxes.

When to Implement

This interface is implemented by object applications in order to provide support for their objects while they are active in-place.

When to Use

These methods are used by the in-place object's top-level container to manipulate objects while they are active.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context-sensitive help.

IOleInPlaceActiveObject Methods	Description
TranslateAccelerator	Translates messages.
OnFrameWindowActivate	State of container's top-level frame.
OnDocWindowActivate	State of container document window.
ResizeBorder	Alert object of need to resize border space.
EnableModeless	Enable or disable modeless dialog boxes.

See Also

IOleWindow

IOleInPlaceActiveObject::EnableModeless Quick Info

Enables or disables modeless dialog boxes when the container creates or destroys a modal dialog box.

```
HRESULT EnableModeless(
  BOOL fEnable      //Enable or disable modeless dialog box windows
);
```

Parameter

fEnable

[in] TRUE to enable modeless dialog box windows; FALSE to disable them.

Return Value

S_OK

The method completed successfully.

Remarks

Notes to Callers

IOleInPlaceActiveObject::EnableModeless is called by the top-level container to enable and disable modeless dialog boxes that the object displays. For the container to display a modal dialog box, it must first call **IOleInPlaceActiveObject::EnableModeless**, specifying FALSE to disable the object's modeless dialog box windows. When the container is through displaying its modal dialog box, it calls **IOleInPlaceActiveObject::EnableModeless**, specifying TRUE to reenable the object's modeless dialog boxes.

See Also

IOleInPlaceFrame::EnableModeless

IOleInPlaceActiveObject::OnDocWindowActivate Quick Info

Notifies the active in-place object when the container's document window is activated or deactivated.

```
HRESULT OnDocWindowActivate(

  BOOL fActivate      //State of MDI child document window
);
```

Parameter

fActivate

[in] State of the MDI child document window. It is TRUE if the window is in the act of activating; FALSE if it is in the act of deactivating.

Return Value

S_OK

The method completed successfully.

Remarks

Notes to Callers

Call **IOleInPlaceActiveObject::OnDocWindowActivate** when the MDI child document window is activated or deactivated and the object is currently the active object for the document.

Notes to Implementers

You should include code in this method that installs frame-level tools during object activation. These tools include the shared composite menu and/or optional toolbars and frame adornments. You should then take focus. When deactivating, the object should remove the frame-level tools. Note that if you do not call **IOleInPlaceUIWindow::SetBorderSpace** with *pborderwidths* set to NULL, you can avoid having to renegotiate border space.

Note While executing **IOleInPlaceActiveObject::OnDocWindowActivate**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **OnDocWindowActivate**.

See Also

PeekMessage, **GetMessage** in Win32

IOleInPlaceActiveObject::OnFrameWindowActivate Quick Info

Notifies the object when the container's top-level frame window is activated or deactivated.

```
HRESULT OnFrameWindowActivate(
  BOOL fActivate      //State of container's top-level window
);
```

Parameter

fActivate

[in] State of the container's top-level frame window. TRUE if the window is activating; FALSE if it is deactivating.

Return Value

S_OK

The method notified the object successfully.

Remarks

Notes to Callers

The container must call **IOleInPlaceActiveObject::OnFrameWindowActivate** when the container's top-level frame window is either being activated or deactivated and the object is the current, active object for the frame.

Note While executing **IOleInPlaceActiveObject::OnFrameWindowActivate**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **OnFrameWindowActivate**.

See Also

PeekMessage, **GetMessage** in Win32

IOleInPlaceActiveObject::ResizeBorder Quick Info

Alerts the object that it needs to resize its border space.

```
HRESULT ResizeBorder(
    LPCRECT prcBorder,              //Pointer to new outer rectangle for border space
    IOleInPlaceUIWindow pUIWindow,  //Pointer to frame or document window border change
    BOOL fFrameWindow               //Indicates whether frame window object calls ResizeBorder
);
```

Parameters

prcBorder

[in] Pointer to a **RECT** structure containing the new outer rectangle within which the object can request border space for its tools.

pUIWindow

[in] Pointer to the frame or document window object whose border has changed.

fFrameWindow

[in] TRUE if the frame window object is calling **ResizeBorder**; otherwise, FALSE.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The method alerted the object successfully.

Remarks

Notes to Callers

IOleInPlaceActiveObject::ResizeBorder is called by the top-level container's document or frame window object when the border space allocated to the object should change. Because the active in-place object is not informed about which window has changed (the frame- or document-level window), **IOleInPlaceActiveObject::ResizeBorder** must be passed the pointer to the window's **IOleInPlaceUIWindow** interface.

Notes to Implementers

In most cases, resizing only requires that you grow, shrink, or scale your object's frame adornments. However, for more complicated adornments, you may be required to renegotiate for border space with calls to **IOleInPlaceUIWindow::RequestBorderSpace** and **IOleInPlaceUIWindow::SetBorderSpace**.

Note While executing **IOleInPlaceActiveObject::ResizeBorder**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **ResizeBorder**.

See Also

IOleInPlaceUIWindow::GetBorder

PeekMessage, **GetMessage** in Win32

IOleInPlaceActiveObject::TranslateAccelerator Quick Info

Processes menu accelerator-key messages from the container's message queue. This method should only be used for objects created by a DLL object application.

```
HRESULT TranslateAccelerator(
  LPMSG lpmsg      //Pointer to message that may need translating
);
```

Parameter

lpmsg

[in] Pointer to the message that might need to be translated.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The message was translated successfully.

S_FALSE

The message was not translated.

Remarks

Notes to Callers

Active in-place objects must always be given the first chance at translating accelerator keystrokes. You can provide this opportunity by calling **IOleInPlaceActiveObject::TranslateAccelerator** from your container's message loop before doing any other translation. You should apply your own translation only when this method returns S_FALSE.

If you call **IOleInPlaceActiveObject::TranslateAccelerator** for an object that is not created by a DLL object application, the default object handler returns S_FALSE.

Notes to Implementers

An object created by an EXE object application gets keystrokes from its own message pump, so the container does not get those messages.

If you need to implement this method, you can do so by simply wrapping the call to the Window's **TranslateAccelerator** function.

See Also

OleTranslateAccelerator

TranslateAccelerator in Win32

IOleInPlaceFrame Quick Info

The **IOleInPlaceFrame** interface controls the container's top-level frame window. This control involves allowing the container to insert its menu group into the composite menu, install the composite menu into the appropriate window frame, and remove the container's menu elements from the composite menu. It sets and displays status text relevant to the in-place object. It also enables or disables the frame's modeless dialog boxes, and translates accelerator keystrokes intended for the container's frame.

When to Implement

You will need to implement this interface if you are writing a container application that will be participating in in-place activation.

When to Use

This interface is used by object applications to control the display and placement of composite menus, keystroke accelerator translation, context-sensitive help mode, and modeless dialog boxes.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context-sensitive help.

IOleInPlaceUIWindow Methods	Description
GetBorder	Translates messages.
RequestBorderSpace	State of container's top-level frame.
SetBorderSpace	State of container document window.
SetActiveObject	Alert object of need to resize border space.

IOleInPlaceFrame Methods	Description
InsertMenus	Allows container to insert menus.
SetMenu	Adds a composite menu to window frame.
RemoveMenus	Removes a container's menu elements.
SetStatusText	Sets and displays status text about.
EnableModeless	Enables or disables modeless dialog boxes.
TranslateAccelerator	Translates keystrokes.

See Also

IOleWindow, IOleInPlaceUIWindow

IOleInPlaceFrame::EnableModeless Quick Info

Enables or disables a frame's modeless dialog boxes.

```
HRESULT EnableModeless(
  BOOL fEnable      //Enable or disable modeless dialog box windows
);
```

Parameter

fEnable

[in] Specifies whether the modeless dialog box windows are to be enabled by specifying TRUE or disabled by specifying FALSE.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The dialog box was either enabled or disabled successfully, depending on the value for *fEnable*.

Remarks

Notes to Callers

The active in-place object calls **IOleInPlaceFrame::EnableModeless** to enable or disable modeless dialog boxes that the container might be displaying. To display a modal dialog box, the object first calls **IOleInPlaceFrame::EnableModeless**, specifying FALSE to disable the container's modeless dialog box windows. After completion, the object calls **IOleInPlaceFrame::EnableModeless,** specifying TRUE to reenable them.

Notes to Implementers

You should track the value of **EnableModeless** and check it before displaying a dialog box.

See Also

IOleInPlaceActiveObject::EnableModeless

IOleInPlaceFrame::InsertMenus Quick Info

Allows the container to insert its menu groups into the composite menu to be used during the in-place session.

```
HRESULT InsertMenus(
  HMENU hmenuShared,                    //Handle to empty menu
  LPOLEMENUGROUPWIDTHS lpMenuWidths     //Pointer to array
);
```

Parameters

hmenuShared

[in] Handle to an empty menu.

lpMenuWidths

[in, out] Pointer to an **OLEMENUGROUPWIDTHS** array of six LONG values. The container fills in elements 0, 2, and 4 to reflect the number of menu elements it provided in the File, View, and Window menu groups.

Return Values

S_OK

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The menu groups were inserted successfully.

Remarks

Notes to Callers

This method is called by object applications when they are first being activated. They call it in order to insert their menus into the frame-level user interface.

The object application asks the container to add its menus to the menu specified in *hmenuShared* and to set the group counts in the **OLEMENUGROUPWIDTHS** array pointed to by *lpMenuWidths*. The object application then adds its own menus and counts. Objects can call **IOleInPlaceFrame::InsertMenus** as many times as necessary to build up the composite menus. The container should use the initial menu handle associated with the composite menu for all menu items in the drop-down menus.

IOleInPlaceFrame::RemoveMenus Quick Info

Gives the container a chance to remove its menu elements from the in-place composite menu.

```
HRESULT RemoveMenus(
  HMENU hmenuShared      //Handle to in-place composite menu
);
```

Parameter

hmenuShared

[in] Handle to the in-place composite menu that was constructed by calls to **IOleInPlaceFrame::InsertMenus** and the Windows **InsertMenu** function.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTEDs, as well as the following:

S_OK

The method completed successfully.

Remarks

The object should always give the container a chance to remove its menu elements from the composite menu before deactivating the shared user interface.

Notes to Callers

Called by the object application while it is being UI-deactivated in order to remove its menus.

See Also

IOleInPlaceFrame::SetMenu

InsertMenu in Win32

IOleInPlaceFrame::SetMenu Quick Info

Installs the composite menu in the window frame containing the object being activated in place.

```
HRESULT SetMenu(
  HMENU hmenuShared,       //Handle to composite menu
  HOLEMENU holemenu,       //Handle to menu descriptor
  HWND hwndActiveObject    //Handle to object's window
);
```

Parameters

hmenuShared

[in] Handle to the composite menu constructed by calls to **IOleInPlaceFrame::InsertMenus** and the Windows **InsertMenu** function.

holemenu

[in] Handle to the menu descriptor returned by the **OleCreateMenuDescriptor** function.

hwndActiveObject

[in] Handle to a window owned by the object and to which menu messages, commands, and accelerators are to be sent.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The method completed successfully.

Remarks

Notes to Callers

The object calls **IOleInPlaceFrame::SetMenu** to ask the container to install the composite menu structure set up by calls to **IOleInPlaceFrame::InsertMenus**.

Notes to Implementers

An SDI container's implementation of this method should call the Windows **SetMenu** function. An MDI container should send a WM_MDISETMENU message, using *hmenuShared* as the menu to install. The container should call **OleSetMenuDescriptor** to install the OLE dispatching code.

When deactivating, the container *must* call **IOleInPlaceFrame::SetMenu**, specifying NULL to remove the shared menu. This is done to help minimize window repaints. The container should also call **OleSetMenuDescriptor**, specifying NULL to unhook the dispatching code. Finally, the object application calls **OleDestroyMenuDescriptor** to free the data structure.

Note While executing **IOleInPlaceFrame::SetMenu**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called

from within **SetMenu**.

See Also

IOleInPlaceFrame::InsertMenus, **OleSetMenuDescriptor**, **OleDestroyMenuDescriptor**

PeekMessage, **GetMessage** in Win32

IOleInPlaceFrame::SetStatusText Quick Info

Sets and displays status text about the in-place object in the container's frame window status line.

```
HRESULT SetStatusText(
  LPCOLESTR pszStatusText      //Pointer to message to display
);
```

Parameter

pszStatusText

[in] Pointer to a null-terminated character string containing the message to display.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_UNEXPECTED, as well as the following:

S_OK

The text was displayed.

S_TRUNCATED

Some text was displayed but the message was too long and was truncated.

Remarks

Notes to Callers

You should call **SetStatusText** when you need to ask the container to display object text in its frame's status line, if it has one. Because the container's frame window owns the status line, calling **IOleInPlaceFrame::SetStatusText** is the *only* way an object can display status information in the container's frame window. If the container refuses the object's request, the object application can, however, negotiate for border space to display its own status window.

Note When switching between menus owned by the container and the in-place active object, the status bar text is not reflected properly if the object does not call the container's **IOleInPlaceFrame::SetStatusText** method. For example, if, during an in-place session, the user were to select the File menu, the status bar would reflect the action that would occur if the user selected this menu. If the user then selects the Edit menu (which is owned by the in-place object), the status bar text would not change unless the **IOleInPlaceFrame::SetStatusText** happened to be called. This is because there is no way for the container to recognize that one of the object's menus has been made active because all the messages that the container would trap are now going to the object.

Notes to Implementers

To avoid potential problems, all objects being activated in place should process the WM_MENUSELECT message and call **IOleInPlaceFrame::SetStatusText** – even if the object does not usually provide status information (in which case the object can just pass a NULL string for the requested status text).

Note While executing **IOleInPlaceFrame::SetStatusText**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **GetBorder**.

See Also

PeekMessage, **GetMessage** in Win32

IOleInPlaceFrame::TranslateAccelerator Quick Info

Translates accelerator keystrokes intended for the container's frame while an object is active in place.

```
HRESULT TranslateAccelerator(
    LPMSG lpmsg,      //Pointer to structure
    WORD wID          //Command identifier value
);
```

Parameters

lpmsg

[in] Pointer to the **MSG** structure containing the keystroke message.

wID

[in] Command identifier value corresponding to the keystroke in the container-provided accelerator table. Containers should use this value instead of translating again.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The keystroke was used.

S_FALSE

The keystroke was not used.

Remarks

Notes to Callers

The **IOleInPlaceFrame::TranslateAccelerator** method is called indirectly by **OleTranslateAccelerator** when a keystroke accelerator intended for the container (frame) is received.

Notes to Implementers

The container application should perform its usual accelerator processing, or use *wID* directly, and then return, indicating whether the keystroke accelerator was processed. If the container is an MDI application and the Windows **TranslateAccelerator** call fails, the container can call the Windows **TranslateMDISysAccel** function, just as it does for its usual message processing.

In-place objects should be given first chance at translating accelerator messages. However, because objects implemented by DLL object applications do not have their own message pump, they receive their messages from the container's message queue. To ensure that the object has first chance at translating messages, a container should always call **IOleInPlaceActiveObject::TranslateAccelerator** before doing its own accelerator translation. Conversely, an executable object application should call **OleTranslateAccelerator** after calling **TranslateAccelerator**, calling **TranslateMessage** and **DispatchMessage** only if both translation functions fail.

Note You should define accelerator tables for containers so they will work properly with object

applications that do their own accelerator keystroke translations. Tables should be defined as follows:

```
"char", wID, VIRTKEY, CONTROL
```

This is the most common way to describe keyboard accelerators. Failure to do so can result in keystrokes being lost or sent to the wrong object during an in-place session.

See Also

OleTranslateAccelerator, **IOleInPlaceActiveObject::TranslateAccelerator**

TranslateAccelerator, **TranslateMessage**, **DispatchMessage**, **TranslateMDISysAccel** in Win32

IOleInPlaceObject Quick Info

The **IOleInPlaceObject** interface manages the activation and deactivation of in-place objects, and determines how much of the in-place object should be visible.

You can obtain a pointer to **IOleInPlaceObject** by calling **QueryInterface** on **IOleObject**.

When to Implement

You must implement this interface if you are writing an object application that will participate in in-place activation.

When to Use

Used by an object's immediate container to activate or deactivate the object.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context sensitive help.

IOleInPlaceObject Methods	Description
InPlaceDeactivate	Deactivate active in-place object.
UIDeactivate	Deactivate and remove UI of active object.
SetObjectRects	Portion of in-place object to be visible.
ReactivateAndUndo	Reactivate previously deactivated object.

See Also

IOleObject, IOleWindow

IOleInPlaceObject::InPlaceDeactivate Quick Info

Deactivates an active in-place object and discards the object's undo state.

HRESULT InPlaceDeactivate();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The object was successfully deactivated.

Remarks

Notes to Callers

This method is called by an active object's immediate container to deactivate the active object and discard its undo state.

Notes to Implementers

On return from **IOleInPlaceObject::InPlaceDeactivate**, the object discards its undo state. The object application should not shut down immediately after this call. Instead, it should wait for an explicit call to **IOleObject::Close** or for the object's reference count to reach zero.

Before deactivating, the object application should give the container a chance to put its user interface back on the frame window by calling **IOleInPlaceSite::OnUIDeactivate**.

If the in-place user interface is still visible during the call to **InPlaceDeactivate**, the object application should call its own **IOleInPlaceObject::UIDeactivate** method to hide the user interface. The in-place user interface can be optionally destroyed during calls to **IOleInPlaceObject::UIDeactivate** and **IOleInPlaceObject::InPlaceDeactivate**. But if the user interface has not already been destroyed when the container calls **IOleObject::Close**, then it *must* be destroyed during the call to **IOleObject::Close**.

During the call to **IOleObject::Close**, the object should check to see whether it is still active in place. If so, it should call **InPlaceDeactivate**.

See Also

IOleInPlaceSite::OnInPlaceDeactivate, IOleInPlaceSite::OnUIDeactivate, IOleObject::Close

IOleInPlaceObject::ReactivateAndUndo Quick Info

Reactivates a previously deactivated object, undoing the last state of the object.

HRESULT ReactivateAndUndo();

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The object was successfully reactivated.

E_NOTUNDOABLE

Called when the Undo state is not available.

Remarks

If the user chooses the Undo command before the Undo state of the object is lost, the object's immediate container calls **IOleInPlaceObject::ReactivateAndUndo** to activate the user interface, carry out the Undo operation, and return the object to the active state.

IOleInPlaceObject::SetObjectRects Quick Info

Indicates how much of the in-place object is visible.

```
HRESULT SetObjectRects(
    LPCRECT lprcPosRect,     //Pointer to the position of the in-place object
    LPCRECT lprcClipRect     //Pointer to the outer rectangle containing the in-place object's position
                             rectangle
);
```

Parameters

lprcPosRect

[in] Pointer to the rectangle containing the position of the in-place object using the client coordinates of its parent window.

lprcClipRect

[in] Pointer to the outer rectangle containing the in-place object's position rectangle (*PosRect*). This rectangle is relative to the client area of the object's parent window.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The operation successfully indicated the rectangle.

Remarks

It is possible for *lprcClipRect* to change without *lprcPosRect* changing.

The size of an in-place object's rectangle is *always* calculated in pixels. This is different from other OLE object's visualizations, which are in HIMETRIC.

Note While executing **IOleInPlaceObject::SetObjectRects**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **SetObjectRects**.

Notes to Callers

The container should call **IOleInPlaceObject::SetObjectRects** whenever the window position of the in-place object and/or the visible part of the in-place object changes.

Notes to Implementers

The object must size its in-place window to match the intersection of *lprcPosRect* and *lprcClipRect*. The object must also draw its contents into the object's in-place window so that proper clipping takes place.

The object should compare its width and height with those provided by its container (conveyed through

lprcPosRect). If the comparison does not result in a match, the container is applying scaling to the object. The object must then decide whether it should continue the in-place editing in the scale/zoom mode or deactivate.

See Also

IOleInPlaceSite::OnPosRectChange

PeekMessage, **GetMessage** in Win32

IOleInPlaceObject::UIDeactivate Quick Info

Deactivates and removes the user interface that supports in-place activation.

HRESULT UIDeactivate();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The in-place UI was deactivated and removed.

Notes to Callers

This method is called by the object's immediate container when, for example, the user has clicked in the client area outside the object.

If the container has called **IOleInPlaceObject::UIDeactivate**, it should later call **IOleInPlaceObject::InPlaceDeactivate** to properly clean up resources. The container can assume that stopping or releasing the object cleans up resources if necessary. The object must be prepared to do so if **IOleInPlaceObject::InPlaceDeactivate** has not been called. but either **IOleInPlaceObject::UIDeactivate** or **IOleObject::Close** has been called.

Notes to Implementers

Resources such as menus and windows can be either cleaned up or kept in a hidden state until your object is completely deactivated by calls to either **IOleInPlaceObject::InPlaceDeactivate** or **IOleObject::Close**. The object application must call **IOleInPlaceSite::OnUIDeactivate** before doing anything with the composite menus so that the container can first be detached from the frame window. On deactivating the in-place object's user interface, the object is left in a ready state so it can be quickly reactivated. The object stays in this state until the undo state of the document changes. The container should then call **IOleInPlaceObject::InPlaceDeactivate** to tell the object to discard its undo state.

See Also

IOleInPlaceObject::InPlaceDeactivate, **IOleInPlaceSite::OnUIDeactivate**, **IOleInPlaceObject::ReactivateAndUndo**, **IOleObject::Close**

IOleInPlaceObjectWindowless Quick Info

The **IOleInPlaceObjectWindowless** interface enables a windowless object to process window messages and participate in drag and drop operations. It is derived from and extends the **IOleInPlaceObject** interface.

A small object, such as a control, does not need a window of its own. Instead, it can rely on its container to dispatch window messages and help the object to participate in drag and drop operations. The container must implement the **IOleInPlaceSiteWindowless** interface. Otherwise, the object must act as a normal compound document object and create a window when it is activated.

When to Implement

Implement this interface on an object that can be in place activated without a window. This interface is derived from **IOleInPlaceObject** which, in turn, is derived from **IOleWindow**.

When to Use

A container calls the methods in this interface to dispatch window messages to an in-place-active windowless object and to assist the object in participating in drag and drop operations. The container must implement a site object with the **IOleInPlaceSiteWindowless** interface to support these activities. See the **IOleInPlaceSiteWindowless** interface for more information on operations involving windowless objects, such as drawing, drag and drop, and processing window messages.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context sensitive help.

IOleInPlaceObject Methods	Description
InPlaceDeactivate	Deactivate active in-place object.
UIDeactivate	Deactivate and remove UI of active object.
SetObjectRects	Portion of in-place object to be visible.
ReactivateAndUndo	Reactivate previously deactivated object.

IOleInPlaceObjectWindowless Methods	Description
OnWindowMessage	Dispatches a message from the container to a windowless object.
GetDropTarget	Supplies the **IDropTarget** interface for a windowless object that

supports drag and drop.

See Also

IOleInPlaceSiteWindowless

IOleInPlaceObjectWindowless::GetDropTarget Quick Info

Supplies the **IDropTarget** interface for an in-place active, windowless object that supports drag and drop.

```
HRESULT GetDropTarget(

  IDropTarget** ppDropTarget     //Indirect pointer to the IDropTarget interface
);
```

Parameters

ppDropTarget

[out] Indirect pointer to the windowless object's **IDropTarget** interface.

Return Values

S_OK

The **IDropTarget** interface was successfully returned.

E_NOTIMPL

The windowless object does not support drag and drop.

Remarks

A windowed object registers its **IDropTarget** interface by calling the **RegisterDragDrop** function and supplying its window handle as a parameter. Registering its **IDropTarget** interface enables the object to participate in drag and drop operations. Because it does not have a window when active, a windowless object cannot register its **IDropTarget** interface. Therefore, it cannot directly participate in drag and drop operations without support from its container.

The following events occur during a drag and drop operation involving windowless objects:

- The container registers its own **IDropTarget** interface through the **RegisterDragDrop** function.
- In the container's implementation of its own **IDropTarget::DragEnter** or **IDropTarget::DragOver** methods, the container detects whether the mouse pointer just entered an embedded object.
- If the object is inactive, the container calls the object's **IPointerInactive::GetActivationPolicy** method. The object returns the POINTERINACTIVE_ACTIVATEONDRAG flag. The container then activates the object in place. If the object was already active, the container does not have to do this step.
- Once the object is active, the container must then obtain the object's **IDropTarget** interface.
- A windowless object that wishes to be a drop target still implements the **IDropTarget** interface, but does not register it and does not return it through calls to **QueryInterface**. Instead, the container can obtain this interface by calling the object's **IOleInPlaceObjectWindowless::GetDropTarget** method. The object returns a pointer to its own **IDropTarget** interface if it wants to participate in drag and drop operations. The container can cache this interface pointer for later use. For example, on subsequent calls to the container's **IDropTarget::DragEnter** or **IDropTarget::DragLeave** methods, the container can use the cached pointer instead of calling the object's **GetDropTarget** method again.
- The container then calls the object's **IDropTarget::DragEnter** and passes the returned value for **pdwEffect* from its own **DragOver** or **DragEnter** methods. From this point on, the container forwards all subsequent **DragOver** calls to the windowless object until the mouse leaves the object or a drop occurs on the object. If the mouse leaves the object, the container calls the object's

IDropTarget::DragLeave and then releases the object's **IDropTarget** interface. If the drop occurs, the container forwards the **IDropTarget::DragDrop** call to the object.

- Finally, the container in-place deactivates the object.

An object can return S_FALSE from its own **IDropTarget::DragEnter** to indicate that it does not accept any of the data formats in the data object. In that case, the container can decide to accept the data for itself and return an appropriate *dwEffect* from its own **DragEnter** or **DragOver** methods. Note that an object that returns S_FALSE from **DragEnter** should be prepared to receive subsequent calls to **DragEnter** without any **DragLeave** in between. Indeed, if the mouse is still over the same object during the next call to the container's **DragOver**, the container may decide to try and call **DragEnter** again on the object.

Note to Callers

A container can cache the pointer to the object's **IDropTarget** interface for later use.

See Also

IDropTarget, **IPointerInactive::GetActivationPolicy**, **RegisterDragDrop**

IOleInPlaceObjectWindowless::OnWindowMessage

Dispatches a message from a container to a windowless object that is in-place active.

```
HRESULT OnWindowMessage(
  UINT msg,                 //Message Identifier as provided by Windows
  WPARAM wParam,            //Message parameter as provided by Windows
  LPARAM lParam,            //Message parameter as provided by Windows
  LRESULT* plResult         //Pointer to message result code
);
```

Parameters

msg

[in] Identifier for the window message provided to the container by Windows.

wParam

[in] Parameter for the window message provided to the container by Windows.

lParam

[in] Parameter for the window message provided to the container by Windows.

plResult

[out] Pointer to result code for the window message as defined in the Windows API.

Return Values

S_OK

The window message was successfully dispatched to the windowless object.

S_FALSE

The windowless object did not process the window message. The container should call the **DefWindowProc** for the message or process the message itself as described below.

Remarks

A container calls this method to send window messages to a windowless object that is in-place active. The container should dispatch messages according to the following guidelines:

- For the following messages, the container should first dispatch the message to the windowless object that has captured the mouse, if any. Otherwise, the container should dispatch the message to the windowless object under the mouse cursor. If there is no such object, the container is free to process the message for itself.

  WM_MOUSEMOVE
  WM_SETCURSOR
  WM_*X*BUTTONDOWN
  WM_*X*BUTTONUP
  WM_*X*BUTTONDBLCLK

- For the following messages, the container should dispatch the message to the windowless object with the keyboard focus.

WM_KEYDOWN	WM_SYSKEYUP
WM_KEYUP	WM_SYSDEADCHAR
WM_CHAR	WM_IME*xxx*
WM_DEADCHAR	WM_HELP
WM_SYSKEYDOWN	WM_CANCELMODE

- For all other messages, the container should process the message on its own.

The windowless object can return S_FALSE to this method to indicate that it did not process the message. Then, the container either performs the default behavior for the message by calling the Windows API function **DefWindowProc**, or processes the message itself.

The container must pass the following window messages to the default window procedure:

WM_MOUSEMOVE	WM_DEADCHAR
WM_*X*BUTTON*xxx*	WM_SYSKEYUP
WM_KEYDOWN	WM_SYSCHAR
WM_KEYUP	WM_SYSDEADCHAR
WM_CHAR	WM_IME*xxx*

The container must process the following window messages as its own:

WM_SETCURSOR
WM_CONTEXTMENU
WM_HELP

Note For WM_SETCURSOR, the container can either set the cursor itself or do nothing.

Objects can also use **IOleInPlaceSiteWindowless::OnDefWindowMessage** to explicitly invoke the default message processing from the container. In the case of the WM_SETCURSOR message, this allows an object to take action if the container does not set the cursor.

All coordinates passed to the object in *wParam* and *lParam* are specified as client coordinates of the containing window.

See Also

IOleInPlaceSiteWindowless::SetCapture, **IOleInPlaceSiteWindowless::OnDefWindowMessage**

IOleInPlaceSite Quick Info

The **IOleInPlaceSite** interface manages interaction between the container and the object's in-place client site. Recall that the client site is the display site for embedded objects, and provides position and conceptual information about the object.

This interface provides methods that manage in-place objects. With **IOleInPlaceSite**, you can determine if an object can be activated and manage its activation and deactivation. You can notify the container when one of its objects is being activated and inform the container that a composite menu will replace the container's regular menu. It provides methods that make it possible for the in-place object to retrieve the window object hierarchy, and the position in the parent window where the object should place its in-place activation window. Finally, it determines how the container scrolls the object, manages the object undo state, and notifies the object when its borders have changed.

When to Implement

You must implement this interface if you are writing a container application that will participate in in-place activation.

When to Use

Use this interface to allow your object to control in-place activation from within the container.

The **IOleInPlaceSite** interface pointer is obtained by calling **QueryInterface** on the object's **IOleClientSite** interface.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context-sensitive help.

IOleInPlaceSite Methods	Description
CanInPlaceActivate	Determines if the container can activate the object in place.
OnInPlaceActivate	Notifies the container that one of its objects is being activated in place.
OnUIActivate	Notifies the container that the object is about to be activated in place, and that the main menu will be replaced by a composite menu.
GetWindowContext	Enables an in-place object to retrieve window interfaces that form at the window object hierarchy, and the position in the

	parent window to locate the object's in-place activation window.
Scroll	Specifies the number of pixels by which the container is to scroll the object.
OnUIDeactivate	Notifies the container to reinstall its user interface and take focus.
OnInPlaceDeactivate	Notifies the container that the object is no longer active in place.
DiscardUndoState	Instructs the container to discard its undo state.
DeactivateAndUndo	Deactivate the object and revert to undo state.
OnPosRectChange	Object's extents have changed.

See Also

IOleWindow, **IOleClientSite**

IOleInPlaceSite::CanInPlaceActivate Quick Info

Determines whether or not the container can activate the object in place.

HRESULT CanInPlaceActivate();

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The container allows in-place activation for this object.

S_FALSE

The container does not allow in-place activation for this object.

Remarks

Only objects being displayed as DVASPECT_CONTENT can be activated in place.

Notes to Callers

IOleInPlaceSite::CanInPlaceActivate is called by the client site's immediate child object when this object must activate in place. This function allows the container application to accept or refuse the activation request.

IOleInPlaceSite::DeactivateAndUndo Quick Info

Causes the container to end the in-place session, deactivate the object, and revert to its own saved undo state.

HRESULT IOleInPlaceSite::DeactivateAndUndo();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The method completed successfully.

Remarks

Notes to Callers

Called by the active object when the user invokes undo just after activating the object.

Notes to Implementers

Upon completion of this call, the container should call **IOleInPlaceObject::UIDeactivate** to remove the user interface for the object, activate itself, and undo.

IOleInPlaceSite::DiscardUndoState Quick Info

Tells the container that the object no longer has any undo state and that the container should not call **IOleInPlaceObject::ReActivateAndUndo**.

HRESULT IOleInPlaceSite::DiscardUndoState();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The method completed successfully.

Remarks

If an object is activated in place and the object's associated object application maintains only one level of undo, there is no need to have more than one entry on the undo stack. That is, once a change has been made to the active object that invalidates its undo state saved by the container, there is no need to maintain this undo state in the container.

Notes to Callers

IOleInPlaceSite::DiscardUndoState is called by the active object while performing some action that would discard the undo state of the object. The in-place object calls this method to notify the container to discard the object's last saved undo state.

See Also

IOleInPlaceSite::DeactivateAndUndo

IOleInPlaceSite::GetWindowContext Quick Info

Enables the in-place object to retrieve the window interfaces that form the window object hierarchy, and the position in the parent window where the object's in-place activation window should be placed.

```
HRESULT GetWindowContext(
  IOleInPlaceFrame **ppFrame,              //Indirect pointer to location of frame interface
  IOleInPlaceUIWindow **ppDoc,             //Indirect pointer to location of document window interface
  LPRECT lprcPosRect,                      //Points to position of in-place object
  LPRECT lprcClipRect,                     //Points to in-place object's position rectangle
  LPOLEINPLACEFRAMEINFO lpFrameInfo        //Points to structure
);
```

Parameters

ppFrame

[out] Indirect pointer to where the **IOleInPlaceFrame** interface on the frame is to be returned. If an error is returned, this parameter must be set to NULL.

ppDoc

[out] Indirect pointer to where the **IOleInPlaceUIWindow** interface on the document window is to be returned. NULL is returned through the *ppDoc* pointer if the document window is the same as the frame window. In this case, the object can only use *ppFrame* or border negotiation. If an error is returned, this parameter must be set to NULL.

lprcPosRect

[out] Pointer to the rectangle containing the position of the in-place object in the client coordinates of its parent window. If an error is returned, this parameter must be set to NULL.

lprcClipRect

[out] Pointer to the outer rectangle containing the in-place object's position rectangle (*PosRect*). This rectangle is relative to the client area of the object's parent window. If an error is returned, this parameter must be set to NULL.

lpFrameInfo

[out] Pointer to an **OLEINPLACEFRAMEINFO** structure the container is to fill in with appropriate data. If an error is returned, this parameter must be set to NULL.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The method completed successfully.

Remarks

The **OLEINPLACEFRAMEINFO** structure provides data needed by OLE to dispatch keystroke accelerators to a container frame while an object is active in place.

When an object is activated, it calls **GetWindowContext** from its container. The container returns the handle to its in-place accelerator table through the **OLEINPLACEFRAMEINFO** structure. Before calling **GetWindowContext**, the object must provide the size of the **OLEINPLACEFRAMEINFO** structure by filling in the *cb* member, pointed to by *lpFrameInfo*.

IOleInPlaceSite::OnInPlaceActivate Quick Info

Notifies the container that one of its objects is being activated in place.

HRESULT OnInPlaceActivate();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The container allows the in-place activation.

Remarks

Notes to Callers

IOleInPlaceSite::OnInPlaceActivate is called by the active embedded object when it is activated in-place for the first time. The container should note that the object is becoming active.

Notes to Implementers

A container that supports linking to embedded objects must properly manage the running of its in-place objects when they are UI-inactive and running in the hidden state. To reactivate the in-place object quickly, a container should *not* call **IOleObject::Close** until the container's **IOleInPlaceSite::DeactivateAndUndo** method is called. To safeguard against the object being left in an unstable state if a linking client updates silently, the container should call **OleLockRunning** to lock the object in the running state. This prevents the hidden in-place object from shutting down before it can be saved in its container.

IOleInPlaceSite::OnInPlaceDeactivate Quick Info

Notifies the container that the object is no longer active in place.

HRESULT OnInPlaceDeactivate();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The method successfully notified the container.

Remarks

Notes to Callers

IOleInPlaceSite::OnInPlaceDeactivate is called by an in-place object when it is fully deactivated. This function notifies the container that the object has been deactivated, and it gives the container a chance to run code pertinent to the object's deactivation. In particular, **IOleInPlaceSite::OnInPlaceDeactivate** is called as a result of **IOleInPlaceObject::InPlaceDeactivate** being called. Calling **IOleInPlaceSite::OnInPlaceDeactivate** indicates that the object can no longer support Undo.

Notes to Implementers

If the container is holding pointers to the **IOleInPlaceObject** and **IOleInPlaceActiveObject** interface implementations, it should release them after the **IOleInPlaceSite::OnInPlaceDeactivate** call.

See Also

IOleInPlaceObject::InPlaceDeactivate

IOleInPlaceSite::OnPosRectChange Quick Info

Indicates the object's extents have changed.

```
HRESULT OnPosRectChange(
  LPCRECT lprcPosRect      //Pointer to rectangle containing the position of in-place object
);
```

Parameter

lprcPosRect

[in] Pointer to the rectangle containing the position of the in-place object in the client coordinates of its parent window.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The method completed successfully.

Remarks

Notes to Callers

The **IOleInPlaceSite::OnPosRectChange** method is called by the in-place object.

Notes to Implementers

When the in-place object calls **IOleInPlaceSite::OnPosRectChange**, the container must call **IOleInPlaceObject::SetObjectRects** to specify the new position of the in-place window and the *ClipRect*. Only then does the object resize its window.

In most cases, the object grows to the right and/or down. There could be cases where the object grows to the left and/or up, as conveyed through *lprcPosRect*. It is also possible to change the object's position without changing its size.

See Also

IOleInPlaceObject::SetObjectRects

IOleInPlaceSite::OnUIActivate Quick Info

Notifies the container that the object is about to be activated in place and that the object is going to replace the container's main menu with an in-place composite menu.

HRESULT IOleInPlaceSite::OnUIActivate();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The container allows the in-place activation.

Remarks

Notes to Callers

The in-place object calls **IOleInPlaceSite::OnUIActivate** just before activating its user interface.

Notes to Implementers

The container should remove any user interface associated with its own activation. If the container is itself an embedded object, it should remove its document-level user interface.

If there is already an object active in place in the same document, the container should call **IOleInPlaceObject::UIDeactivate** before calling **OnUIDeactivate**.

See Also

IOleInPlaceObject::UIDeactivate

IOleInPlaceSite::OnUIDeactivate Quick Info

Notifies the container on deactivation that it should reinstall its user interface and take focus, and whether or not the object has an undoable state.

```
HRESULT OnUIDeactivate(
  BOOL fUndoable      //Specifies if object can undo changes
);
```

Parameter

fUndoable

[in] Specifies whether the object can undo changes. TRUE if the object can undo, FALSE if it cannot.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The method completed successfully.

Remarks

The object indicates whether it can undo changes through the *fUndoable* flag. If the object can undo changes, the container can (by the user invoking the Edit Undo command) call the **IOleInPlaceObject::ReactivateAndUndo** method to undo the changes.

Notes to Callers

IOleInPlaceSite::OnUIDeactivate is called by the site's immediate child object when it is deactivating to notify the container that it should reinstall its own user interface components and take focus. The container should wait for the call to **IOleInPlaceSite::OnUIDeactivate** to complete before fully cleaning up and destroying any composite submenus.

See Also

IOleInPlaceObject::ReactivateAndUndo

IOleInPlaceSite::Scroll Quick Info

Tells the container to scroll the view of the object by a specified number of pixels.

```
HRESULT Scroll(
  SIZE scrollExtent      //Number of pixels
);
```

Parameter

scrollExtent

[in] Number of pixels by which to scroll in the *X* and *Y* directions.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The method successfully executed the view scroll instruction.

Remarks

As a result of scrolling, the object's visible rectangle can change. If that happens, the container should give the new *ClipRect* to the object by calling **IOleInPlaceObject::SetObjectRects**. The intersection of the *ClipRect* and *PosRect* rectangles gives the new visible rectangle. See **IOleInPlaceSite::GetWindowContext** for a discussion of *ClipRect* and *PosRect*.

Notes to Callers

Called by an active, in-place object when it is asking the container to scroll.

See Also

IOleInPlaceObject::SetObjectRects

IOleInPlaceSiteEx Quick Info

The **IOleInPlaceSiteEx** interface provides an additional set of activation and deactivation notification methods that enable an object to avoid unnecessary flashing on the screen when the object is activated and deactivated.

When an object is activated, it does not know if its visual display is already correct. When the object is deactivated, the container does not know if the visual display is correct. To avoid a redraw and the associated screen flicker in both cases, the container can provide this extension to **IOleInPlaceSite**.

When to Implement

Implement this interface on the client site in a container application that supports flicker-free activation and deactivation of embedded, in-place active objects.

When to Use

An embedded object, such as a control, calls the methods in this interface to determine if it needs to redraw itself on activation and to notify the container if the object needs to be redrawn on deactivation. By avoiding the redraw when it is not needed, you can reduce the amount of flashing on the screen.

If the site object does not support **IOleInPlaceSiteEx**, the object must call methods in the **IOleInPlaceSite** interface instead. In this case, the object must redraw itself on activation and deactivation.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context-sensitive help.

IOleInPlaceSite Methods	Description
CanInPlaceActivate	Determines if the container can activate the object in place.
OnInPlaceActivate	Notifies the container that one of its objects is being activated in place.
OnUIActivate	Notifies the container that the object is about to be activated in place, and that the main menu will be replaced by a composite menu.
GetWindowContext	Enables an in-place object to retrieve window interfaces that form at the window object hierarchy, and the position in the parent window to locate the

	object's in-place activation window.
Scroll	Specifies the number of pixels by which the container is to scroll the object.
OnUIDeactivate	Notifies the container to reinstall its user interface and take focus.
OnInPlaceDeactivate	Notifies the container that the object is no longer active in place.
DiscardUndoState	Instructs the container to discard its undo state.
DeactivateAndUndo	Deactivate the object and revert to undo state.
OnPosRectChange	Object's extents have changed.

IOleInPlaceSiteEx Methods	**Description**
OnInPlaceActivateEx	Called by the embedded object to determine if it needs to redraw itself upon activation.
OnInPlaceDeactivateEx	Notifies the container of whether the object needs to be redrawn upon deactivation.
RequestUIActivate	Notifies the container that the object is about to enter the UI-active state.

See Also

IOleInPlaceSite

IOleInPlaceSiteEx::OnInPlaceActivateEx Quick Info

Called by the embedded object to determine if it needs to redraw itself upon activation.

```
HRESULT OnInPlaceActivateEx(
  BOOL* pfNoRedraw,        //Pointer to current redraw status
  DWORD dwFlags            //Indicates whether the object is windowless
);
```

Parameters

pfNoRedraw

[out] Pointer to current redraw status. The status is TRUE if the object need not redraw itself upon activation; FALSE if the object needs to redraw upon activation. Windowless objects usually do not need the value returned by this parameter and may pass a NULL pointer to save the container the burden of computing this value.

dwFlags

[in] Indicates whether the object is activated as a windowless object. This parameter takes values from the **ACTIVATEFLAGS** enumeration. See **IOleInPlaceSiteWindowless** for more information on windowless objects.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The container allows the in-place activation.

Remarks

This method replaces **IOleInPlaceSite::OnInPlaceActivate**. If the older method is used, the object must always redraw itself on activation.

Windowless objects are required to use this method instead of **IOleInPlaceSite::OnInPlaceActivate** to notify the container of whether they are activating windowless or not.

Notes to Implementers

The container should carefully check the invalidation status of the object, its z-order, clipping and any other relevant parameters to determine the appropriate value to return in *pfNoRedraw*.

A container can cache the value of the **ACTIVATEFLAGS** enumeration instead of calling the **GetWindow** method in the **IOleInPlaceObjectWindowless** interface repeatedly.

See Also

ACTIVATEFLAGS, **IOleInPlaceSite::OnInPlaceActivate**, **IOleInPlaceObjectWindowless**
IOleInPlaceSiteWindowless

IOleInPlaceSiteEx::OnInPlaceDeactivateEx Quick Info

Notifies the container if the object needs to be redrawn upon deactivation.

```
HRESULT OnInPlaceDeactivateEx(
  BOOL fNoRedraw              //Indicates whether the object needs to be redrawn
);
```

Parameter

fNoRedraw

[in] If TRUE, the container need not redraw the object after completing the deactivation; if FALSE the object must be redrawn after deactivation.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The method successfully notified the container.

Remarks

This method replaces **IOleInPlaceSite::OnInPlaceDeactivate**. If the older method is used, the object must always be redrawn on deactivation.

See Also

IOleInPlaceSite::OnInPlaceDeactivate

IOleInPlaceSiteEx::RequestUIActivate Quick Info

Notifies the container that the object is about to enter the UI-active state.

HRESULT RequestUIActivate(void);

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The object can continue the activation process and call **IOleInPlaceSite::OnUIActivate**.

S_FALSE

The object cannot enter the UI-active state. The object must call **IOleInPlaceSite::OnUIDeactivate** so the container can perform its the necessary processing to restore the focus.

Remarks

An object calls this method to determine if it can enter the UI-active state and to notify the container that it is about to make this transition. The container can return S_FALSE to deny this request, for example, if the end user has canceled the operation or if the currently active object will not relinquish its active state.

If the object does not call **RequestUIActivate**, the container handles data validation and fires Enter and Exit events from **IOleInPlaceSite::OnUIActivate**.

See Also

IOleInPlaceSite::OnUIActivate, **IOleInPlaceSite::OnUIDeactivate**

IOleInPlaceSiteWindowless Quick Info

The **IOleInPlaceSiteWindowless** interface is derived from and extends the **IOleInPlaceSiteEx** interface. **IOleInPlaceSiteWindowless** works with **IOleInPlaceObjectWindowless** which is implemented on the windowless object. Together, these two interfaces provide services to a windowless object from its container allowing the windowless object to:

- process window messages
- participate in drag and drop operations
- perform drawing operations

Having a window can place unnecessary burdens on small objects, such as controls. It prevents an object from being non-rectangular. It prevents windows from being transparent. It prevents the small instance size needed by many small controls.

A windowless object can enter the in-place active state without requiring a window or the resources associated with a window. Instead, the object's container provides the object with many of the services associated with having a window.

Windowless object model

Windowless objects are an extension of normal compound document objects. They follow the same in-place activation model and share the same definitions for the various OLE states, with the difference that they do not consume a window when they enter the in-place and UI active states. They are required to comply with the OLE compound document specification, including in-place and UI activation.

Windowless objects require special support from their container. In other words, the container has to be specifically written to support this new kind of object. However, windowless objects are backward compatible with down level containers. In such containers, they simply create a window when active and behave as a normal compound document object.

As with other compound document objects, windowless objects need to be in-place active to get mouse and keyboard messages. In fact, since an object needs to have the keyboard focus to receive keyboard messages, and having the keyboard focus implies being UI active for an object, only UI active objects will actually get keyboard messages. Non-active objects can still process keyboard mnemonics.

Since windowless objects do not have a window, they rely on their container to receive window messages for them. The container dispatches its own window messages to the appropriate embedded, windowless object through calls to **IOleInPlaceObjectWindowless** methods. Similarly, windowless objects can obtain services from their container such as capturing the mouse, setting the focus, getting a device context in which to paint, and so on. The control calls **IOleInPlaceSiteWindowless** methods. In addition, the container is responsible for drawing any border hatching as well as the grab handles for the control.

These two interfaces are derived from existing interfaces. By extending existing interfaces rather than creating new ones, no new VTable pointer is added to the object instance, helping to keep the instance size small.

Client and server negotiations with windowless objects

When a windowless object gets in-place activated, it should query its site for the **IOleInPlaceSiteWindowless** interface. If this interface is supported, the object calls **IOleInPlaceSiteWindowless::CanWindowlessActivate** to determine if it can proceed and in-place activate without a window.

If the container does not support **IOleInPlaceSiteWindowless** or if the **IOleInPlaceSiteWindowless::CanWindowlessActivate** method returns S_FALSE, the windowless

object should behave like a normal compound document object and create a window.

The container can get the window handle for an embedded object by calling **IOleWindow::GetWindow**. This method should fail (return E_FAIL) for a windowless object. However, a container cannot be sure that the object is windowless by calling this method. The object may have a window but may not have created it yet. Many existing objects only create their window after calling the **OnInPlaceActivate** method on their site object.

Consequently, a windowless object must call the new **IOleInPlaceSiteEx::OnInPlaceActivateEx** method on its site object, instead of **OnInPlaceActivate**. The *dwFlags* parameter for this new method contains additional information in the ACTIVATEFLAGS enumeration. The ACTIVATE_WINDOWLESS enumeration value indicates that the object is activated without a window. Containers can cache this value instead of calling the **GetWindow** method on the **IOleWindow** interface repeatedly.

Message dispatching

Windowless objects rely on their containers to dispatch window messages to them, capture the mouse, and get the keyboard focus. The container calls **IOleInPlaceObject::OnWindowMessage** to dispatch a window message to a windowless object. This method is similar to the **SendMessage** Windows API function except that it does not require an HWND parameter and it returns both an HRESULT and a LRESULT.

A windowless object must not call the **DefWindowProc** Windows API function directly. Instead, it calls **IOleInPlaceSiteWindowless::OnDefWindowMessage** to invoke the default action, for example with WM_SETCURSOR or WM_HELP that should be propagated back up to the container. Thus, the container has a chance to handle the message before the object processes it.

For mouse messages, the object calls **IOleInPlaceSiteWindowless::Set Capture** to obtain the mouse capture and **IOleInPlaceSiteWindowless::SetFocus** to get the keyboard focus.

A windowless object handles accelerators and mnemonics as follows:

Accelerators

The UI active object checks for its own accelerators in **IOleInPlaceActiveObject::TranslateAccelerator**. A windowless object does the same. However, a windowless object cannot send a WM_COMMAND message to itself, as a windowed object would do. Therefore, instead of translating the key to a command, a windowless object should simply process the key right away.

Except for that one difference, windowless objects should implement the **IOleInPlaceActiveObject::TranslateAccelerator** method as defined in the OLE specifications. In particular, a windowless object should pass the accelerator message up to its site object if it does not wish to handle it. The windowless object and returns S_OK if the message got translated and S_FALSE if not. In the case of a windowless object, the message is processed instead of translated. Non translated messages will come back to the object through the **IOleInPlaceObjectWindowless::OnWindowMessage** method.

Note that because the container's window gets all keyboard input, a UI active object should look for messages sent to that window to find those it needs to process in its **IOleInPlaceActiveObject::TranslateAccelerator** method. An object can get its container's window by calling **IOleWindow::GetWindow**.

Mnemonics

Control mnemonics are handled the same way whether the control is windowless or not. The container gets the control mnemonic table by calling **IOleControl::GetControlInfo** and then calls **IOleControl::OnMnemonic** when it receives a key combination that matches a control mnemonic.

Drag & drop onto windowless objects

Since a windowless object does not have a window when it is active, it cannot register its own **IDropTarget** interface with the **RegisterDragDrop** function. However, to participate in drag and drop operations, a windowless object still must implement this interface. The object supplies its container with a pointer to its **IDropTarget** interface through **IOleInPlaceObjectWindowless::GetDropTarget** instead of having the container call **QueryInterface** for it. See **IOleInPlaceObjectWindowless::GetDropTarget** for more information on drag and drop operations involving windowless objects.

In-place drawing for windowless objects

With windowed objects, the container is only responsible for drawing the object when it is inactive. Windowed objects have their own window when they are active and can draw themselves independently of their container.

A windowless object, however, needs services from its container to redraw itself even when it is active. The container must provide the object with information about its surrounding environment, such as the clipping region, the background, overlapping objects in front of the object being redrawn, and a device context in which to draw.

The **IOleInPlaceSiteWindowless** interface on the container's site object provides these services: drawing the object, obtaining and releasing the device context, invalidating the object's on-screen display, scrolling the object, or showing a caret when the object is active.

> **Note** All methods of **IOleInPlaceSiteWindowless** take position information in client coordinates of the containing window, that is, the window in which the object is being drawn.

Drawing windowless objects

To maintain compatibility with windowed objects, the container still uses **IViewObject::Draw** to redraw the in-place active, windowless object. See **IViewObject::Draw** for information on how the method is used with windowless objects.

Obtaining and releasing a device context

To draw on its own when in-place active, a windowless object must call its site's **IOleInPlaceSiteWindowless::GetDC** method to get a device context in which to draw. Then, it draws into the device context and releases it by calling **IOleInPlaceSiteWindowless::ReleaseDC**.

Display Invalidation

In-place windowless objects may need to invalidate regions of their on-screen image. Even though the notification methods in **IAdviseSinkEx** can be used for that purpose, they are not ideal for in-place active objects because they take HIMETRIC coordinates. In order to simplify and speed up in-place drawing, the **InvalidateRect** and **InvalidateRgn** methods in the **IOleInPlaceSiteWindowless** interface provide the same functionality.

An object cannot call the Windows API functions **InvalidateRect** and **InvalidateRgn** directly on the window handle it gets from calling **IOleInPlaceSiteWindowless::GetWindow** on its site.

Scrolling

In-place active windowless objects may need to scroll a given rectangle of their on-screen image, for example, with a multi-line text control. Because of transparent and overlapping objects, the Windows API functions **ScrollWindow** and **ScrollDC** cannot be used. Instead, the **IOleInPlaceSiteWindowless::ScrollRect** method enables objects to perform scrolling.

Caret support

A windowless object cannot safely show a caret without first checking whether the caret is partially or totally hidden by overlapping objects. In order to make that possible, an object can submit a rectangle to its site object when it calls the site's **IOleInPlaceSiteWindowless::AdjustRect** method to get a

specified rectangle adjusted (usually, reduced) to ensure the rectangle fits in the clipping region.

When to Implement

Implement this interface on the container's site object to support windowless objects.

When to Use

The windowless object calls the methods in this interface to process window messages, to participate in drag and drop operations, and to perform drawing operations.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context-sensitive help.

IOleInPlaceSite Methods	Description
CanInPlaceActivate	Determines if the container can activate the object in place.
OnInPlaceActivate	Notifies the container that one of its objects is being activated in place.
OnUIActivate	Notifies the container that the object is about to be activated in place, and that the main menu will be replaced by a composite menu.
GetWindowContext	Enables an in-place object to retrieve window interfaces that form at the window object hierarchy, and the position in the parent window to locate the object's in-place activation window.
Scroll	Specifies the number of pixels by which the container is to scroll the object.
OnUIDeactivate	Notifies the container to reinstall its user interface and take focus.
OnInPlaceDeactivate	Notifies the container that the object is no longer active in place.
DiscardUndoState	Instructs the container to discard its undo state.
DeactivateAndUndo	Deactivate the object and revert to undo state.
OnPosRectChange	Object's extents have changed.

IOleInPlaceSiteEx Methods	Description
OnInPlaceActivateEx	Called by the embedded object to determine if it needs to redraw itself upon activation.
OnInPlaceDeactivateEx	Notifies the container of whether the object needs to be redrawn upon deactivation.
RequestUIActivate	Notifies the container that the object is about to enter the UI-active state.

IOleInPlaceSiteWindowless Methods	Description
CanWindowlessActivate	Informs an object if its container can support it as a windowless object that can be in-place activated.
GetCapture	Called by an in-place active, windowless object to determine if it still has the mouse capture or not.
SetCapture	Enables an in-place active, windowless object to capture all mouse messages.
GetFocus	Called by an in-place active, windowless object to determine if it still has the keyboard focus or not.
SetFocus	Sets the keyboard focus for a UI-active, windowless object.
GetDC	Provides an object with a handle to a device context for a screen or compatible device from its container.
ReleaseDC	Releases the device context previously obtained by a call to **IOleInPlaceSiteWindowless::GetDC**.
InvalidateRect	Enables an object to invalidate a specified rectangle of its in-place image on the screen.
InvalidateRgn	Enables an object to invalidate a specified region of its in-place image on the screen.
ScrollRect	Enables an object to scroll an area within its in-place active image on the screen.
AdjustRect	Adjusts a specified rectangle if it is entirely or partially covered by overlapping, opaque objects.
OnDefWindowMessage	Invokes the default processing for all messages passed to an object.

See Also

IAdviseSinkEx, IOleControl, IOleInPlaceActiveObject::TranslateAccelerator, IOleInPlaceObjectWindowless,

IOleInPlaceSiteWindowless::AdjustRect Quick Info

Adjusts a specified rectangle if it is entirely or partially covered by overlapping, opaque objects.

```
HRESULT AdjustRect(
  LPRECT prc                    //Rectangle to adjust
);
```

Parameters

prc

[in,out] Rectangle to adjust.

Return Values

S_OK

The rectangle was adjusted successfully. Note S_OK means that the rectangle was not completely covered.

S_FALSE

The rectangle was adjusted successfully. Note S_FALSE means that the rectangle was completely covered. Its width and height are now NULL.

Remarks

The main use of this method is to adjust the size of the caret. An object willing to create a caret should submit the caret rectangle to its site object by calling this method and using the adjusted rectangle returned from it for the caret. If the caret is entirely hidden, this method will return S_FALSE and the caret should not be shown at all in this case.

In a situation where objects are overlapping this method should return the largest rectangle that is fully visible.

This method can also be used to figure whether a point or a rectangular area is visible or hidden by overlapping objects.

IOleInPlaceSiteWindowless::CanWindowlessActivate Quick Info

Informs an object if its container can support it as a windowless object that can be in-place activated.

HRESULT CanWindowlessActivate(void);

Return Values

S_OK

The object can activate in place without a window.

Remarks

If this method returns S_OK, the container can dispatch events to it using **IOleInPlaceObjectWindowless**.

If this method returns S_FALSE, the object should create a window and behave as a normal compound document object.

See Also

IOleInPlaceObjectWindowless

IOleInPlaceSiteWindowless::GetCapture Quick Info

Called by an in-place active, windowless object to determine if it still has the mouse capture or not.

HRESULT GetCapture(void);

Return Values

S_OK

The object currently has the mouse capture.

S_FALSE

The object does not currently have the mouse capture.

Remarks

As an alternative to calling this method, the object can cache information about whether it has the mouse capture or not.

IOleInPlaceSiteWindowless::GetDC Quick Info

Provides an object with a handle to a device context for a screen or compatible device from its container.

```
HRESULT GetDC(
  LPCRECT pRect,          //Pointer to rectangle
  DWORD grfFlags,         //OLEDCFLAGS value
  HDC* phDC               //Pointer to device context
);
```

Parameters

pRect

[in] Pointer to the rectangle that the object wants to redraw, in client coordinates of the containing window. If this parameter is NULL, the object's full extent is redrawn.

grfFlags

[in] A combination of values from the **OLEDCFLAGS** enumeration.

phDC

[out] Pointer to a returned device context.

Return Values

S_OK

A device context was successfully returned.

OLE_E_NESTEDPAINT

The container is already in the middle of a paint session. That is, this method has already been called, and the **ReleaseDC** method has not yet been called.

Remarks

A device context obtained by this method should be released by calling **IOleInPlaceSiteWindowless::ReleaseDC**.

Like other methods in this interface, rectangles are specified in client coordinates of the containing window. The container is expected to intersect this rectangle with the object's site rectangle and clip out everything outside the resulting rectangle. This prevents objects from inadvertently drawing where they are not supposed to.

Containers are also expected to map the device context origin so the object can draw in client coordinates of the containing window, usually the container's window. If the container is merely passing its window device context, this occurs automatically. If it is returning another device context, for example, an offscreen memory device context, then the viewport origin should be set appropriately.

Notes to Implementers

Depending whether it is returning an on-screen or off-screen device context and depending on how sophisticated it is, container can use one of the following algorithms:

1. **On-screen, one pass drawing**

In the **GetDC** method, the container should:

- Get the window device context.
- If OLEDC_PAINTBKGND is set, draw the DVASPECT_CONTENT aspect of every object behind the object requesting the device context.
- Return the device context.

In the **ReleaseDC** method, the container should:

- Draw the DVASPECT_CONTENT of every overlapping object.
- Release the device context.

2. **On-screen, two pass drawing**

   In the **GetDC** method, the container should:

   - Get the window device context.
   - Clip out the opaque regions of any overlapping object. These regions do not need to be redrawn since they are already correct on the screen.
   - If OLEDC_PAINTBKGND is not set, return the device context.
   - Otherwise, clip out the opaque parts of the object requesting the device context and draw the opaque parts of every object behind it going front to back.
   - Draw the transparent aspects of every object behind going back to front, setting the clipping region appropriately each time.
   - Finally return the device context.

   In the **ReleaseDC** method, the container should:

   - Draw the transparent parts of every overlapping object.
   - Release the device context.

3. **Off-screen drawing**

   In the **GetDC** method, the container should:

   - Create a screen compatible memory device context, containing a compatible bitmap of appropriate size.
   - Map the viewport origin of the device context to ensure that the calling object can draw using client area coordinates of the containing window.
   - If OLEDC_PAINTBKGND is set, draw the DVASPECT_CONTENT of every object behind the calling object.
   - Return the device context.

   In the **ReleaseDC** method, the container should:

   - Draw the DVASPECT_CONTENT aspect of every overlapping object.
   - Copy the off-screen bitmap to the screen at the location the calling object originally requested in **GetDC**.
   - Delete and release the memory device context.

When this method returns, the clipping region in the device context should be set so that the object can't paint in any area it is not supposed to. If the object is not opaque, the background should have been painted. If the device context is a screen, any overlapping opaque areas should be clipped out.

See Also

IOleInPlaceSiteWindowless::ReleaseDC, OLEDCFLAGS

IOleInPlaceSiteWindowless::GetFocus

Called by an in-place active, windowless object to determine if it still has the keyboard focus or not.

HRESULT GetFocus(void);

Return Values

S_OK

The object currently has the keyboard focus.

S_FALSE

The object does not currently have the keyboard focus.

Remarks

A windowless object calls this method to find out if it currently has the focus or not. As an alternative to calling this method, the object can cache information about whether it has the keyboard focus or not.

IOleInPlaceSiteWindowless::InvalidateRect Quick Info

Enables an object to invalidate a specified rectangle of its in-place image on the screen.

```
HRESULT InvalidateRect(
  LPCRECT pRect,      //Rectangle to be invalidated
  BOOL fErase         //Indicates whether to erase the background
);
```

Parameters

pRect

[in] Rectangle to invalidate, in client coordinates of the containing window. If this parameter is NULL, the object's full extent is invalidated.

fErase

[in] Specifies whether the background within the update region is to be erased when the region is updated. If this parameter is TRUE, the background is erased. If this parameter is FALSE, the background remains unchanged.

Return Values

S_OK

The specified rectangle was successfully invalidated.

Remarks

An object is only allowed to invalidate pixels contained in its own site rectangle. Any attempt to invalidate an area outside of that rectangle should result in a no-op.

IOleInPlaceSiteWindowless::InvalidateRgn Quick Info

Enables an object to invalidate a specified region of its in-place image on the screen.

```
HRESULT InvalidateRgn(
  HRGN hRGN,          //Region to be invalidated
  BOOL fErase         //Indicates whether to erase the background
);
```

Parameters

hRGN

[in] Region to invalidate, in client coordinates of the containing window. If this parameter is NULL, the object's full extent is invalidated.

fErase

[in] Specifies whether the background within the update region is to be erased when the region is updated. If this parameter is TRUE, the background is erased. If this parameter is FALSE, the background remains unchanged.

Return Values

S_OK

The specified region was successfully invalidated.

Remarks

An object is only allowed to invalidate pixels contained in its own site region. Any attempt to invalidate an area outside of that region should result in a no-op.

IOleInPlaceSiteWindowless::OnDefWindowMessage

Quick Info

Invokes the default processing for all messages passed to an object.

```
HRESULT OnDefWindowMessage(
  UINT msg,                 //Message Identifier as provided by Windows
  WPARAM wParam,            //Message parameter as provided by Windows
  LPARAM lParam,            //Message parameter as provided by Windows
  LRESULT* plResult         //Pointer to message result code
);
```

Parameters

msg

[in] Identifier for the window message provided to the container by Windows.

wParam

[in] Parameter for the window message provided to the container by Windows.

lParam

[in] Parameter for the window message provided to the container by Windows.

plResult

[out] Pointer to result code for the window message as defined in the Windows API.

Return Values

S_OK

The container's default processing for the window message was successfully invoked.

S_FALSE

The container's default processing for the window message was not invoked. See Note to Implementers below.

Remarks

A windowless object can explicitly invoke the default processing for a window message by calling this method. A container dispatches window messages to its windowless objects by calling **IOleInPlaceObjectWindowless::OnWindowMessage**. The object usually returns S_FALSE to indicate that it did not process the message. Then, the container can perform the default behavior for the message by calling the Windows API function **DefWindowProc**.

Instead, the object can call this method on the container's site object to explicitly invoke the default processing. Then, the object can take action on its own if the container does not handle the message.

Note to Implementers

The container must pass the following window messages to its default window procedure (the **DefWindowProc** Windows API function) and return S_OK. Note that **plResult* should contain the value returned by **DefWindowProc**.

WM_MOUSEMOVE	WM_DEADCHAR
WM_*X*BUTTON*xxx*	WM_SYSKEYUP
WM_KEYDOWN	WM_SYSCHAR
WM_KEYUP	WM_SYSDEADCHAR
WM_CHAR	WM_IME*xxx*

The container can either process the window messages as its own and return S_OK or not do anything and return S_FALSE.

WM_SETCURSOR
WM_CONTEXTMENU
WM_HELP

If the container returns S_FALSE, the object can take action to process the window message on its own.

See Also

IOleInPlaceObjectWindowless::OnWindowMessage

IOleInPlaceSiteWindowless::ReleaseDC Quick Info

Releases the device context previously obtained by a call to **IOleInPlaceSiteWindowless::GetDC**.

```
HRESULT ReleaseDC(
  HDC hDC                    //Device context to be released
);
```

Parameters

hDC

[in] Specifies the device context to be released.

Return Values

S_OK

The device context was successfully released.

Remarks

An object calls this method to notify its container that the object is done with the device context. If the device context was used for drawing, the container should ensure that all overlapping objects are repainted correctly. If the device context was an offscreen device context, its content should also be copied to the screen in the rectangle originally passed to **IOleInPlaceSiteWindowless::GetDC**. See **IOleInPlaceSiteWindowless::GetDC** for implementation notes relevant to **ReleaseDC**.

See Also

IOleInPlaceSiteWindowless::GetDC

IOleInPlaceSiteWindowless::ScrollRect Quick Info

Enables an object to scroll an area within its in-place active image on the screen.

```
HRESULT ScrollRect(
  int dx,                  //Amount to scroll on the x-axis
  int dy,                  //Amount to scroll on the y-axis
  LPCRECT pRectScroll,     //Rectangle to scroll
  LPCRECT pRectClip        //Rectangle to clip
);
```

Parameters

dx

[in] Amount to scroll on the x-axis.

dy

[in] Amount to scroll on the y-axis.

pRectScroll

[in] Rectangle to scroll, in client coordinates of the containing window. NULL means the full object.

pRectClip

[in] Rectangle to clip to as defined for the Windows API function. Only pixels scrolling into this rectangle are drawn. Pixels scrolling out are not. If this parameter is NULL, the rectangle is not clipped.

Return Values

S_OK

The rectangle was successfully scrolled.

Remarks

This method should take in account the fact that the caller may be transparent and that there may be opaque or transparent overlapping objects. See Notes to Implementers below for suggestions on algorithms this method can use.

Note to Implementers

Containers can implement this method in a variety of ways. However, all of them should account for the possibility that the object requesting scrolling may be transparent or may not have a solid background. Containers should also take into account that there may be overlapping objects.

The simplest way to implement this method consists in simply redrawing the rectangle to scroll.

An added refinement to this simple implementation is to use the **ScrollDC** Windows API function when the object requesting the scroll is opaque, the object has a solid background, and there are no overlapping objects.

More sophisticated implementations can use the following procedure:

- Check whether the object is opaque and has a solid background, using **IViewObjectEx::GetViewStatus**. If not, simply invalidate the rectangle to scroll. An added refinement is to check whether the scrolling rectangle is entirely in the opaque region of a partially transparent object.
- Get the window device context.
- Clip out the opaque parts of any overlapping object returned by **IViewObjectEx::GetRect**.
- Clip out and invalidate the transparent parts of any overlapping object.
- Finally, call the Windows API function **ScrollDC**.
- Redraw the previously invalidated transparent parts of any overlapping object.

Regardless of the scrolling and clipping rectangle, only pixels contained in the object's site rectangle will be painted. The area uncovered by the scrolling operation is invalidated and redrawn immediately, before this method returns.

All redraw generated by this method should happen synchronously before this method returns.

This method should automatically hide the caret during the scrolling operation and should move the caret by the scrolling amounts if it is inside the clip rectangle.

See Also

IViewObjectEx::GetViewStatus, **IViewObjectEx::GetRect**

IOleInPlaceSiteWindowless::SetCapture Quick Info

Enables an in-place active, windowless object to capture all mouse messages.

```
HRESULT SetCapture(
  BOOL fCapture                 //Set or release mouse capture
);
```

Parameters

fCapture

[in] If TRUE, the container should capture the mouse for the object. If FALSE, the container should release mouse capture for the object.

Return Values

S_OK

Mouse capture was successfully granted to the object. If called to release the mouse capture, this method must not fail.

S_FALSE

Mouse capture was denied to the object.

Remarks

A windowless object captures the mouse input, by calling **IOleInPlaceSiteWindowless::SetCapture(TRUE)** on its site object. The container can deny mouse capture, in which case this method returns S_FALSE. If the capture is granted, the container must set the Windows mouse capture to its own window and dispatch any subsequent mouse message to the object, regardless of whether the mouse cursor position is over this object or not.

The object can later release mouse capture by calling **IOleInPlaceSiteWindowless::SetCapture(FALSE)** on its site object. The capture can also be released because of an external event, such as the ESC key being pressed. In this case, the object is notified by a WM_CANCELMODE message that the container dispatches along with the keyboard focus.

Containers should dispatch all mouse messages, including WM_SETCURSOR, to the windowless OLE object that has captured the mouse. If no object has captured the mouse, the container should dispatch the mouse message to the object under the mouse cursor.

The container dispatches these window messages by calling **IOleInPlaceObjectWindowless::OnWindowMessage** on the windowless object. The windowless object can return S_FALSE to this method to indicate that it did not process the mouse message. Then, the container should perform the default behavior for the message by calling the Windows API function **DefWindowProc**. For WM_SETCURSOR, the container can either set the cursor itself or do nothing.

Objects can also use **IOleInPlaceSiteWindowless::OnDefWindowMessage** to invoke the default message processing from the container. In the case of the WM_SETCURSOR message, this allows an object to take action if the container does not set the cursor.

See Also

IOleInPlaceSiteWindowless::OnDefWindowMessage

IOleInPlaceSiteWindowless::SetFocus Quick Info

Sets the keyboard focus for a UI-active, windowless object.

```
HRESULT SetFocus(
  BOOL fFocus                    //Set or release the keyboard focus
);
```

Parameters

fFocus

[in] If TRUE, sets the keyboard focus to the calling object. If FALSE, removes the keyboard focus from the calling object, provided that the object has the focus.

Return Values

S_OK

Keyboard focus was successfully given to the object. If this method is called to release the focus, it should never fail.

S_FALSE

Keyboard focus was denied to the object.

Remarks

A windowless object calls this method whenever a windowed object would call the Windows API function **SetFocus**. Through this call, the windowless object obtains the keyboard focus and can respond to window messages. Normally, this call is made during the UI activation process and within the notification methods **IOleInPlaceActiveObject::OnDocWindowActivate(TRUE)** and **IOleInPlaceActiveObject::OnFrameWindowActivate(TRUE)**.

In response to this call, the container sets the Windows focus to the window being used to get keyboard messages (usually the container window) and redirects any subsequent keyboard messages to the windowless object that requested the focus.

A windowless object also calls the **IOleInPlaceSiteWindowless::SetFocus** method with the *fFocus* parameter set to FALSE to release the keyboard focus without assigning it to any other object. In this case, the container must call the Windows API function **SetFocus** with a NULL parameter so that no window has the focus.

See Also

IOleInPlaceActiveObject

IOleInPlaceUIWindow Quick Info

The **IOleInPlaceUIWindow** interface is implemented by container applications and used by object applications to negotiate border space on the document or frame window. The container provides a **RECT** structure in which the object can place toolbars and other similar controls, determines if tools can in fact be installed around the object's window frame, allocates space for the border, and establishes a communication channel between the object and each frame and document window.

The document window may not exist in all applications. When this is the case, **IOleInPlaceSite::GetWindowContext** returns NULL for **IOleInPlaceUIWindow**.

When to Implement

You must implement this interface if you are writing a container application that will participate in in-place activation.

When to Use

Used by object applications to negotiate border space on the document or frame window when one of its objects is being activated, or to renegotiate border space if the size of the object changes.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context-sensitive help.

IOleInPlaceUIWindow Methods	Description
GetBorder	Specifies a RECT structure for toolbars and controls.
RequestBorderSpace	Determines if tools can be installed around object's window frame.
SetBorderSpace	Allocates space for the border.
SetActiveObject	Provides for direct communication between the object and each document and frame window.

See Also

IOleWindow

IOleInPlaceUIWindow::GetBorder Quick Info

Returns a **RECT** structure in which the object can put toolbars and similar controls while active in place.

```
HRESULT GetBorder(
  LPRECT lprectBorder      //Pointer to structure
);
```

Parameter

lprectBorder

[out] Pointer to a **RECT** structure where the outer rectangle is to be returned. The **RECT** structure's coordinates are relative to the window being represented by the interface.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The rectangle was successfully returned.

E_NOTOOLSPACE

The object cannot install toolbars in this window object.

Remarks

Notes to Callers

The **IOleInPlaceUIWindow::GetBorder** function, when called on a document or frame window object, returns the outer rectangle (relative to the window) where the object can put toolbars or similar controls.

If the object is to install these tools, it should negotiate space for the tools within this rectangle using **IOleInPlaceUIWindow::RequestBorderSpace** and then call **IOleInPlaceUIWindow::SetBorderSpace** to get this space allocated.

Note While executing **IOleInPlaceUIWindow::GetBorder**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **GetBorder**.

See Also

IOleInPlaceUIWindow::RequestBorderSpace, **IOleInPlaceUIWindow::SetBorderSpace**

PeekMessage, **GetMessage** in Win32

IOleInPlaceUIWindow::RequestBorderSpace Quick Info

Determines if there is available space for tools to be installed around the object's window frame while the object is active in place.

```
HRESULT RequestBorderSpace(
    LPCBORDERWIDTHS pborderwidths      //Pointer to a structure
);
```

Parameter

pborderwidths

[in] Pointer to a **BORDERWIDTHS** structure containing the requested widths (in pixels) needed on each side of the window for the tools.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The requested space could be allocated to the object.

E_NOTOOLSPACE

The object cannot install toolbars in this window object because the implementation does not support toolbars, or there is insufficient space to install the toolbars.

Remarks

Notes to Callers

The active in-place object calls **IOleInPlaceUIWindow::RequestBorderSpace** to ask if tools can be installed inside the window frame. These tools would be allocated between the rectangle returned by **IOleInPlaceUIWindow::GetBorder** and the **BORDERWIDTHS** structure specified in the argument to this call.

The space for the tools is not actually allocated to the object until it calls **IOleInPlaceUIWindow::SetBorderSpace**, allowing the object to negotiate for space (such as while dragging toolbars around), but deferring the moving of tools until the action is completed.

The object can install these tools by passing the width in pixels that is to be used on each side. For example, if the object required 10 pixels on the top, 0 pixels on the bottom, and 5 pixels on the left and right sides, it would pass the following **BORDERWIDTHS** structure to **IOleInPlaceUIWindow::RequestBorderSpace**:

```
lpbw->top    = 10
lpbw->bottom =  0
lpbw->lLeft  =  5
lpbw->right  =  5
```

Note While executing **IOleInPlaceUIWindow::RequestBorderSpace**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system

to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **RequestBorderSpace**.

Notes to Implementers

If the amount of space an active object uses for its toolbars is irrelevant to the container, it can simply return NOERROR as shown in the following **IOleInPlaceUIWindow::RequestBorderSpace** example. Containers should not unduly restrict the display of tools by an active in-place object.

```
HRESULT InPlaceUIWindow_RequestBorderSpace(
    IOleInPlaceFrame *  lpThis,
    LPCBORDERWIDTHS     pborderwidths)
{
    /* Container allows the object to have as much border space as it
    ** wants.
    */
    return NOERROR;
}
```

See Also

IOleInPlaceUIWindow::GetBorder, **IOleInPlaceUIWindow::SetBorderSpace** **PeekMessage**, **GetMessage** in Win32

IOleInPlaceUIWindow::SetActiveObject Quick Info

Provides a direct channel of communication between the object and each of the frame and document windows.

```
HRESULT SetActiveObject(
  IOleInPlaceActiveObject *pActiveObject,    //Pointer to active in-place object
  LPCOLESTR pszObjName                       //Pointer tostring containing a name describing the object
);
```

Parameters

pActiveObject

[in] Pointer to the **IOleInPlaceActiveObject** interface on the active in-place object.

pszObjName

[in] Pointer to a string containing a name that describes the object an embedding container can use in composing its window title. It can be NULL if the object does not require the container to change its window titles. The *Microsoft Windows User Interface Design Guide* recommends that containers ignore this parameter and always use their own name in the title bar.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The method completed successfully.

Remarks

Generally, an embedded object should pass NULL for the *pszObjName* parameter (see "Notes to Implementers" below). However, if you are working in conjunction with a container that does display the name of the in-place active object in its title bar, then you should compose a string in the following form:

<application name> - <object short-type name>

Notes to Callers

IOleInPlaceUIWindow::SetActiveObject is called by the object to establish a direct communication link between itself and the document and frame windows.

When deactivating, the object calls **IOleInPlaceUIWindow::SetActiveObject**, passing NULL for the *pActiveObject* and *pszObjName* parameters.

An object *must* call **IOleInPlaceUIWindow::SetActiveObject** before calling **IOleInPlaceFrame::SetMenu** to give the container the pointer to the active object. The container then uses this pointer in processing **IOleInPlaceFrame::SetMenu** and to pass to **OleSetMenuDescriptor**.

Notes to Implementers

The *Microsoft Windows User Interface Design Guide* recommends that an in-place container ignore the *pszObjName* parameter passed in this method. The guide says "The title bar is not affected by in-place

activation. It always displays the top-level container's name."

See Also

IOleInPlaceFrame::SetMenu, **OleSetMenuDescriptor**

IOleInPlaceUIWindow::SetBorderSpace Quick Info

Allocates space for the border requested in the call to **IOleInPlaceUIWindow::RequestBorderSpace**.

```
HRESULT SetBorderSpace(
  LPCBORDERWIDTHS pborderwidths      //Pointer to a structure
);
```

Parameter

pborderwidths

[in] Pointer to a **BORDERWIDTHS** structure containing the requested width (in pixels) of the tools. It can be NULL, indicating the object does not need any space.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The requested space has been allocated to the object.

OLE_E_INVALIDRECT

The rectangle does not lie within the specifications returned by **IOleInPlaceUIWindow::GetBorder**.

Remarks

The object must call **IOleInPlaceUIWindow::SetBorderSpace**. It can do any one of the following:

- Use its own toolbars, requesting border space of a specific size, or,
- Use no toolbars, but force the container to remove its toolbars by passing a valid **BORDERWIDTHS** structure containing nothing but zeros in the *pborderwidths* parameter, or,
- Use no toolbars but allow the in-place container to leave its toolbars up by passing NULL as the *pborderwidths* parameter.

The **BORDERWIDTHS** structure used in this call would generally have been passed in a previous call to **IOleInPlaceUIWindow::RequestBorderSpace**, which must have returned S_OK.

If an object must renegotiate space on the border, it can call **SetBorderSpace** again with the new widths. If the call to **SetBorderSpace** fails, the object can do a full negotiation for border space with calls to **GetBorder**, **RequestBorderSpace**, and **SetBorderSpace**.

Note While executing **IOleInPlaceUIWindow::SetBorderSpace**, do not make calls to the Windows **PeekMessage** or **GetMessage** functions, or a dialog box. Doing so may cause the system to deadlock. There are further restrictions on which OLE interface methods and functions can be called from within **SetBorderSpace**.

See Also

IOleInPlaceUIWindow::GetBorder, **IOleInPlaceUIWindow::RequestBorderSpace** **PeekMessage**, **GetMessage** in Win32

IOleItemContainer Quick Info

The **IOleItemContainer** interface is used by item monikers when they are bound to the objects they identify.

When any container of objects uses item monikers to identify its objects, it must define a naming scheme for those objects. The container's **IOleItemContainer** implementation uses knowledge of that naming scheme to retrieve an object given a particular name. Item monikers use the container's **IOleItemContainer** implementation during binding.

When to Implement

You must implement **IOleItemContainer** if you're a moniker provider handing out item monikers. Being a moniker provider means handing out monikers that identify your objects to make them accessible to moniker clients. You must use item monikers if the objects you're identifying are contained within another object and can be individually identified using a string.

The most common example of moniker providers are OLE applications that support linking. If your OLE application supports linking to objects smaller than a file-based document, you need to use item monikers. For a server application that allows linking to a portion of a document (such as selections within a document), you use the item monikers to identify those objects. For a container application that allows linking to embedded objects, you use the item monikers to identify the embedded objects.

You must define a naming scheme for identifying the objects within the container; for example, embedded objects in a document could be identified with names of the form "embedobj1," "embedobj2," and so forth, while ranges of cells in a spreadsheet could be identified with names of the form "A1:E7," "G5:M9," and so forth. (Ranges of cells in a spreadsheet are examples of "pseudo-objects" because they do not have their own persistent storage, but simply represent a portion of the container's internal state.) You create an item moniker that represents an object's name using the **CreateItemMoniker** function and hand it out to a moniker client. When an item moniker is bound, your implementation of **IOleItemContainer** must be able to take a name and retrieve the corresponding object.

When to Use

Applications typically do not call **IOleItemContainer** methods directly. The item moniker implementation of **IMoniker** is the primary caller of **IOleItemContainer** methods.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IParseDisplayName Method	Description
ParseDisplayName	Parses object's display name to form moniker.

IOleContainer Methods	Description
EnumObjects	Enumerates objects in a container.
LockContainer	Keeps container running until explicitly released.

IOleItemContainer Methods	Description
GetObject	Returns a pointer to a specified object.
GetObjectStorage	Returns a pointer to an object's storage.
IsRunning	Checks whether an object is running.

See Also

CreateItemMoniker, IMoniker - Item Moniker Implementation

IOleItemContainer::GetObject Quick Info

Returns a pointer to the object identified by the specified name.

```
HRESULT GetObject(
  LPOLESTR pszItem,          //Pointer to name of the object requested
  DWORD dwSpeedNeeded,       //Speed requirements on binding
  IBindCtx *pbc,             //Pointer to bind context object to be used
  REFIID riid,               //Reference to the identifier of the interface pointer desired
  void **ppvObject           //Indirect pointer to interface
);
```

Parameters

pszItem

[in] Pointer to a zero-terminated string containing the container's name for the requested object. For Win32 applications, the **LPOLESTR** type indicates a wide character string (two bytes per character); otherwise, the string has one byte per character.

dwSpeedNeeded

[in] Indicates approximately how long the caller will wait to get the object. The legal values for *dwSpeedNeeded* are taken from the enumeration **BINDSPEED**. For information on the **BINDSPEED** enumeration, see the "Data Structures" section.

pbc

[in] Pointer to the **IBindCtx** interface on the bind context object to be used in this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the binding implementation should retrieve information about its environment. For more information, see **IBindCtx**.

riid

[in] Reference to the identifier of the interface pointer requested.

ppvObject

[out] When successful, indirect pointer to the location of the interface specified in *riid* on the object named by *pszItem*. In this case, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs, the implementation sets *ppvObject* to NULL.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The specified object was successfully returned.

MK_E_EXCEEDEDDEADLINE

The binding operation could not be completed within the time limit specified by the bind context's **BIND_OPTS** structure, or with the speed indicated by the *dwSpeedNeeded* parameter.

MK_E_NOOBJECT

The parameter *pszItem* does not identify an object in this container.

E_NOINTERFACE

The requested interface was not available.

Remarks

The item moniker implementation of **IMoniker::BindToObject** calls this method, passing the name stored within the item moniker as the *pszItem* parameter.

Notes to Implementers

Your implementation of **IOleItemContainer::GetObject** should first determine whether *pszItem* is a valid name for one of the container's objects. If not, you should return MK_E_NOOBJECT.

If *pszItem* names an embedded or linked object, your implementation must check the value of the *dwSpeedNeeded* parameter. If the value is BINDSPEED_IMMEDIATE and the object is not yet loaded, you should return MK_E_EXCEEDEDDEADLINE. If the object is loaded, your implementation should determine whether the object is running (for example, by calling the **OleIsRunning** function). If it is not running and the *dwSpeedNeeded* value is BINDSPEED_MODERATE, your implementation should return MK_E_EXCEEDEDDEADLINE. If the object is not running and *dwSpeedNeeded* is BINDSPEED_INDEFINITE, your implementation should call the **OleRun** function to put the object in the running state. Then it can query the object for the requested interface. Note that it is important the object be running before you query for the interface.

If *pszItem* names a pseudo-object, your implementation can ignore the *dwSpeedNeeded* parameter because a pseudo-object is running whenever its container is running. In this case, your implementation can simply query for the requested interface.

If you want more specific information about the time limit than is given by *dwSpeedNeeded*, you can call **IBindCtx::GetBindOptions** on the *pbc* parameter to get the actual deadline parameter.

See Also

IMoniker::BindToObject, **IBindCtx::GetBindOptions**, **OleIsRunning**, **OleRun**

IOleItemContainer::GetObjectStorage Quick Info

Returns a pointer to the storage for the object identified by the specified name.

```
HRESULT GetObjectStorage(
    LPOLESTR pszItem,       //Name of the string containing the name of object whose storage is requested
    IBindCtx *pbc,          //Pointer to bind context to be used
    REFIID riid,            //Reference to the identifier of the interface pointer desired
    void **ppvStorage       //Indirect pointer to object's storage
);
```

Parameters

pszItem

[in] Pointer to a zero-terminated string containing the compound document's name for the object whose storage is requested. For Win32 applications, the **LPOLESTR** type indicates a wide character string (two bytes per character); otherwise, the string has one byte per character.

pbc

[in] Pointer to the **IBindCtx** interface on the bind context to be used in this binding operation. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the binding implementation should retrieve information about its environment. For more information, see **IBindCtx**.

riid

[in] Reference to the identifier of the interface to be used to communicate with the object, usually **IStorage**.

ppvStorage

[out] When successful, indirect pointer to the location of the interface specified in *riid*, on the storage for the object named by *pszItem*. In this case, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs, *ppvStorage* is set to NULL.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The storage of the specified object was successfully returned.

MK_E_NOOBJECT

The parameter *pszItem* does not identify a object in this container.

MK_E_NOSTORAGE

The object does not have its own independent storage.

E_NOINTERFACE

The requested interface is not available.

Remarks

The item moniker implementation of **IMoniker::BindToStorage** calls this method.

Notes to Implementers

If *pszItem* designates a pseudo-object, your implementation should return MK_E_NOSTORAGE, because pseudo-objects do not have their own independent storage. If *pszItem* designates an embedded object, or a portion of the document that has its own storage, your implementation should return the specified interface pointer on the appropriate storage object.

See Also

IMoniker - Item Moniker Implementation

IOleItemContainer::IsRunning Quick Info

Indicates whether the object identified by the specified name is running.

```
HRESULT IsRunning(
  LPOLESTR pszItem      //Pointer to string containing name of object
);
```

Parameter

pszItem

[in] Pointer to a zero-terminated wide character string (two bytes per character) containing the container's name for the object.

Return Values

S_OK

The specified object is running.

S_FALSE

The object is not running.

MK_E_NOOBJECT

The parameter *pszItem* does not identify an object in this container.

Remarks

The item moniker implementation of **IMoniker::IsRunning** calls this method.

Notes to Implementers

Your implementation of **IOleItemContainer::IsRunning** should first determine whether *pszItem* identifies one of the container's objects. If it does not, your implementation should return MK_E_NOOBJECT. If the object is not loaded, your implementation should return S_FALSE. If it is loaded, your implementation can call the **OleIsRunning** function to determine whether it is running.

If *pszItem* names a pseudo-object, your implementation can simply return S_OK because a pseudo-object is running whenever its container is running.

See Also

IMoniker::IsRunning

IOleLink Quick Info

The **IOleLink** interface is the means by which a linked object provides its container with functions pertaining to linking. The most important of these functions is binding to the link source, that is, activating the connection to the document that stores the linked object's native data. **IOleLink** also defines functions for managing information about the linked object, such as the location of the link source and the cached presentation data for the linked object.

A container application can distinguish between embedded objects and linked objects by querying for **IOleLink**; only linked objects implement **IOleLink**.

When to Implement

You do not have to implement this interface yourself; the system supplies an implementation of **IOleLink** that is suitable for all situations. This implementation is used automatically whenever you create or load a linked object.

When to Use

You must use **IOleLink** if you are writing a container application that allows its documents to contain linked objects. You primarily call **IOleLink** methods in order to implement the Links dialog box. If you use the **OleUIEditLinks** function to display the Links dialog box, your calls to **IOleLink** methods take place in your implementation of the **IOleUILinkContainer** interface.

Some **IOleLink** methods don't have to be called directly. Instead, you call methods of **IOleObject**; the default linked object provides an implementation of **IOleObject** that often calls methods of **IOleLink**. For example, a container application typically activates a linked object by calling **IOleObject::DoVerb**, which in turn calls **IOleLink::BindToSource**.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleLink Methods	Description
SetUpdateOptions	Sets the update options.
GetUpdateOptions	Returns the update options.
SetSourceMoniker	Sets the moniker for the link source.
GetSourceMoniker	Returns the moniker for the link source.
SetSourceDisplayName	Sets the display name for the link source.
GetSourceDisplayName	Returns the display name for the link source.
BindToSource	Binds the moniker to the link source.
BindIfRunning	Binds the moniker if the source is running.
GetBoundSource	Returns a pointer to the link source if it's running.

UnbindSource	Break connection to the link source.
Update	Update the cached views of the link source.

See Also

IOleObject, **IOleUILinkContainer**, **OleUIEditLinks**

IOleLink::BindIfRunning Quick Info

Activates the connection between the linked object and the link source if the link source is already running.

HRESULT BindIfRunning(*void***);**

Return Values

S_OK

The link source was bound.

S_FALSE

The link source is not running.

CreateBindCtx, **IMoniker::IsRunning**, or **IOleLink::BindToSource** errors

Binding the moniker might require calling these functions, therefore, errors generated by these functions may be returned.

Remarks

You typically do not need to call **IOleLink::BindIfRunning**. This method is primarily called by the linked object.

Notes on Provided Implementation

The linked object's implementation of **IOleLink::BindIfRunning** checks the Running Object Table (ROT) to determine whether the link source is already running. It checks both the relative and absolute monikers. If the link source is running, **IOleLink::BindIfRunning** calls **IOleLink::BindToSource** to connect the linked object to the link source.

See Also

IOleLink::BindToSource

IOleLink::BindToSource Quick Info

Activates the connection to the link source by binding the moniker stored within the linked object.

```
HRESULT BindToSource(
    DWORD bindflags,     //Flag in case CLSID of link source is different
    IBindCtx *pbc        //Pointer to bind context to be used
);
```

Parameters

bindflags

[in] Specifies how to proceed if the link source has a different CLSID from the last time it was bound. If this parameter is zero and the CLSIDs are different, the method fails and returns OLE_E_CLASSDIFF. If the OLELINKBIND_EVENIFCLASSDIFF value from the **OLELINKBIND** enumeration is specified and the CLSIDs are different, the method binds successfully and updates the CLSID stored in the linked object.

pbc

[in] Pointer to the **IBindCtx** interface on the bind context to be used in this binding operation. This parameter can be NULL. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the binding implementation should retrieve information about its environment. For more information, see **IBindCtx**.

Return Values

S_OK

The link source is bound.

OLE_E_CLASSDIFF

The link source was not bound because its CLSID has changed. This error is returned only if the OLELINKBIND_EVENIFCLASSDIFF flag is not specified in the *bindflags* parameter.

MK_E_NOOBJECT

The link source could not be found or (if the link source's moniker is a composite) some intermediate object identified in the composite could not be found.

E_UNSPEC

The link's moniker is NULL.

CreateBindCtx errors

Binding the moniker might require calling this function; therefore, this method may return errors generated by this function.

Remarks

Notes to Callers

Typically, your container application does not need to call the **IOleLink::BindToSource** method directly. When it's necessary to activate the connection to the link source, your container typically calls

IOleObject::DoVerb, **IOleObject::Update**, or **IOleLink::Update**. The linked object's implementation of these methods calls **IOleLink::BindToSource**. Your container can also call the **OleRun** function, which – when called on a linked object – calls **IOleLink::BindToSource**.

In each of the examples listed previously, in which **IOleLink::BindToSource** is called indirectly, the *bindflags* parameter is set to zero. Consequently, these calls can fail with the OLE_E_CLASSDIFF error if the class of the link source is different from what it was the last time the linked object was bound. This could happen, for example, if the original link source was an embedded Lotus spreadsheet that an end user had subsequently converted (using the Change Type dialog box) to an Excel spreadsheet.

If you want your container to bind even though the link source now has a different CLSID, you can call **IOleLink::BindToSource** directly and specify OLELINKBIND_EVENIFCLASSDIFF for the *bindflags* parameter. This call binds to the link source and updates the link object's CLSID. Alternatively, your container can delete the existing link and use the **OleCreateLink** function to create a new linked object.

Notes on Provided Implementation

The linked object caches the interface pointer to the link source acquired during binding.

The linked object's **IOleLink::BindToSource** implementation first tries to bind using a moniker consisting of the compound document's moniker composed with the link source's relative moniker. If successful, it updates the link's absolute moniker. Otherwise, it tries to bind using the absolute moniker, updating the relative moniker if successful.

If **IOleLink::BindToSource** binds to the link source, it calls the compound document's **IOleContainer::LockContainer** implementation to keep the containing compound document alive while the link source is running. **IOleLink::BindToSource** also calls the **IOleObject::Advise** and **IDataObject::DAdvise** implementations of the link source to set up advisory connections. The **IOleLink::UnbindSource** implementation unlocks the container and deletes the advisory connections.

See Also

IDataObject::DAdvise, **IOleContainer::LockContainer**, **IOleLink::Update**, **IOleLink::UnbindSource**, **IOleObject::Advise**, **IOleObject::DoVerb**, **IOleObject::Update**, **OleRun**

IOleLink::GetBoundSource Quick Info

Returns an **IUnknown** pointer to the link source if the connection is currently active.

```
HRESULT GetBoundSource(
  IUnknown **ppunk      //Indirect pointer to location of the link source
);
```

Parameter

ppunk

[out] When successful, indirect pointer to the location of the **IUnknown** interface on the link source. In this case, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs, the implementation sets *ppunk* to NULL.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

A pointer was returned successfully.

Remarks

You typically do not need to call **IOleLink::GetBoundSource**.

IOleLink::GetSourceDisplayName Quick Info

Retrieves the display name of the link source of the linked object.

```
HRESULT GetSourceDisplayName(
  LPOLESTR *ppszDisplayName      //Indirect pointer to string containing display name of link source
);
```

Parameter

ppszDisplayName

[out] Indirect pointer to the location of a zero-terminated wide character string (two bytes per character) containing the display name of the link source. If an error occurs, *ppszDisplayName* is set to NULL; otherwise, the implementation must use **IMalloc::Alloc** to allocate the string returned in *ppszDisplayName*, and the caller is responsible for calling **IMalloc::Free** to free it. Both caller and called use the allocator returned by **CoGetMalloc**.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The display name was successfully retrieved.

CreateBindCtx and **IMoniker::GetDisplayName** errors

Retrieving the display name requires calling these functions; therefore, this method may return errors generated by these functions.

Remarks

Notes to Callers

Your container application can call **IOleLink::GetSourceDisplayName** in order to display the current source of a link.

The current source of a link is displayed in the Links dialog box. If you use the **OleUIEditLinks** function to display the Links dialog box, you must implement the **IOleUILinkContainer** interface. The dialog box calls your implementations of **IOleUILinkContainer::GetLinkSource** to get the string it should display. Your implementation of that method can call **IOleLink::GetSourceDisplayName**.

Notes on Provided Implementation

The linked object's implementation of **IOleLink::GetSourceDisplayName** calls **IOleLink::GetSourceMoniker** to get the link source moniker, and then calls **IMoniker::GetDisplayName** to get that moniker's display name. This operation is potentially expensive because it might require binding the moniker. All of the system-supplied monikers can return a display name without binding, but there is no guarantee that other moniker implementations can. Instead of making repeated calls to **IOleLink::GetSourceDisplayName**, your container application can cache the name and update it whenever the link source is bound.

See Also

IOleLink::SetSourceDisplayName, **IOleUILinkContainer**, **IMoniker::GetDisplayName**,

OleUIEditLinks

IOleLink::GetSourceMoniker Quick Info

Retrieves the moniker identifying the link source of a linked object.

```
HRESULT GetSourceMoniker(
  IMoniker **ppmk      //Indirect pointer to a moniker identifying link source
);
```

Parameter

ppmk

[out] When successful, indirect pointer to the **IMoniker** interface on an absolute moniker that identifies the link source. In this case, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. May be NULL if an error occurs.

Return Values

S_OK

The moniker was returned successfully.

MK_E_UNAVAILABLE

No moniker is available.

Remarks

Notes to Callers

Your container application can call **IOleLink::GetSourceMoniker** to display the current source of a link in the Links dialog box. Note that this requires your container to use the **IMoniker::GetDisplayName** method to get the display name of the moniker. If you'd rather get the display name directly, your container can call **IOleLink::GetSourceDisplayName** instead of **IOleLink::GetSourceMoniker**.

If you use the **OleUIEditLinks** function to display the Links dialog box, you must implement the **IOleUILinkContainer** interface. The dialog box calls your implementations of **IOleUILinkContainer::GetLinkSource** to get the string it should display. Your implementation of that method can call **IOleLink::GetSourceMoniker**.

Notes on Provided Implementation

The linked object stores both an absolute and a relative moniker for the link source. If the relative moniker is non-NULL and a moniker is available for the compound document, **IOleLink::GetSourceMoniker** returns the moniker created by composing the relative moniker onto the end of the compound document's moniker. Otherwise, it returns the absolute moniker or, if an error occurs, NULL.

The container specifies the absolute moniker when it calls one of the **OleCreateLink** functions to create a link. The application can call **IOleLink::SetSourceMoniker** or **IOleLink::SetSourceDisplayName** to change the absolute moniker. In addition, the linked object automatically updates the monikers whenever it successfully binds to the link source, or when it is bound to the link source and it receives a rename notification through the **IAdviseSink::OnRename** method.

See Also

IOleLink::SetSourceDisplayName, **IOleLink::SetSourceMoniker**

IOleLink::GetUpdateOptions Quick Info

Retrieves a value indicating how often the linked object updates its cached data.

```
HRESULT GetUpdateOptions(

  DWORD *pdwUpdateOpt      //Pointer to update option
);
```

Parameter

pdwUpdateOpt

[out] Pointer to a DWORD that specifies the current value for the linked object's update option, indicating how often the linked object updates the cached data for the linked object. The legal values for *pdwUpdateOpt* are taken from the enumeration **OLEUPDATE**.

Return Value

S_OK

The update option was retrieved successfully.

Remarks

Notes to Callers

Your container application should call **IOleLink::GetUpdateOptions** to display the current update option for a linked object.

A linked object's current update option is displayed in the Links dialog box. If you use the **OleUIEditLinks** function to display the Links dialog box, you must implement the **IOleUILinkContainer** interface. The dialog box calls your implementation of **IOleUILinkContainer::GetLinkUpdateOptions** to determine which update option it should display. Your implementation of that method should call **IOleLink::GetUpdateOptions** to retrieve the current update option.

See Also

IOleLink::SetUpdateOptions, **IOleUILinkContainer**, **OleUIEditLinks**

IOleLink::SetSourceDisplayName Quick Info

Specifies the new link source of a linked object using a display name.

```
HRESULT SetSourceDisplayName(
  LPCOLESTR pszStatusText      //Pointer to display name of new link source
);
```

Parameter

pszStatusText

[in] Pointer to the display name of the new link source. It may not be NULL.

Return Values

S_OK

The display name was set successfully.

MkParseDisplayName errors

Setting the display name requires calling this function; therefore, this method may return errors generated by this function.

Remarks

Notes to Callers

Your container application can call **IOleLink::SetSourceDisplayName** when the end user changes the source of a link or breaks a link. Note that this requires the linked object to create a moniker out of the display name. If you'd rather parse the display name into a moniker yourself, your container can call **IOleLink::SetSourceMoniker** instead of **IOleLink::SetSourceDisplayName**.

If you use the **OleUIEditLinks** function to display the Links dialog box, you must implement the **IOleUILinkContainer** interface. The dialog box calls your implementations of **IOleUILinkContainer::SetLinkSource** and **IOleUILinkContainer::CancelLink**. Your implementation of these methods can call **IOleLink::SetSourceDisplayName**.

If your container application is immediately going to bind to a newly specified link source, you should call **MkParseDisplayName** and **IOleLink::SetSourceMoniker** instead, and then call **IOleLink::BindToSource** using the bind context from the parsing operation. By reusing the bind context, you can avoid redundant loading of objects that might otherwise occur.

Notes on Provided Implementation

The contract for **IOleLink::SetSourceDisplayName** does not specify when the linked object will parse the display name into a moniker. The parsing can occur before **IOleLink::SetSourceDisplayName** returns, or the linked object can store the display name and parse it only when it needs to bind to the link source. Note that parsing the display name is potentially an expensive operation because it might require binding to the link source. The provided implementation of **IOleLink::SetSourceDisplayName** parses the display name and then releases the bind context used in the parse operation. This can result in running and then stopping the link source server.

If the linked object is bound to the current link source, the implementation of **IOleLink::SetSourceDisplayName** breaks the connection.

For more information on how the linked object stores and uses the moniker to the link source, see **IOleLink::SetSourceMoniker**.

See Also

IOleLink::SetSourceMoniker, **IOleUILinkContainer**, **MkParseDisplayName**, **OleUIEditLinks**

IOleLink::SetSourceMoniker Quick Info

Specifies the new link source of a linked object using a moniker.

```
HRESULT SetSourceMoniker(
    IMoniker *pmk,      //Pointer to a moniker identifying new link source
    REFCLSID rclsid     //CLSID of link source
);
```

Parameters

pmk

[in] Pointer to the **IMoniker** interface on a moniker that identifies the new link source of the linked object. A value of NULL breaks the link.

rclsid

[in] Specifies the CLSID of the link source that the linked object should use to access information about the linked object when it is not bound.

Return Value

S_OK

The moniker was set successfully.

Remarks

Notes to Callers

Your container application can call **IOleLink::SetSourceMoniker** when the end user changes the source of a link or breaks a link. Note that this requires your container to use the **MkParseDisplayName** function to create a moniker out of the display name that the end user enters. If you'd rather have the linked object perform the parsing, your container can call **IOleLink::SetSourceDisplayName** instead of **IOleLink::SetSourceMoniker**.

The end user changes the source of a link or breaks a link using the Links dialog box. If you use the **OleUIEditLinks** function to display the Links dialog box, you must implement the **IOleUILinkContainer** interface. The dialog box calls your implementations of **IOleUILinkContainer::SetLinkSource** and **IOleUILinkContainer::CancelLink**; your implementation of these methods can call **IOleLink::SetSourceMoniker**.

If the linked object is currently bound to its link source, the linked object's implementation of **IOleLink::SetSourceMoniker** closes the link before changing the moniker.

Notes on Provided Implementation

The **IOleLink** contract does not specify how the linked object stores or uses the link source moniker. The provided implementation stores the absolute moniker specified when the link is created or when the moniker is changed; it then computes and stores a relative moniker. Future implementations might manage monikers differently to provide better moniker tracking. The absolute moniker provides the complete path to the link source. The linked object uses this absolute moniker and the moniker of the compound document to compute a relative moniker that identifies the link source relative to the compound document that contains the link.

```
pmkCompoundDoc->RelativePathTo(pmkAbsolute, ppmkRelative)
```

When binding to the link source, the linked object first tries to bind using the relative moniker. If that fails, it tries to bind the absolute moniker.

When the linked object successfully binds using either the relative or the absolute moniker, it automatically updates the other moniker. The linked object also updates both monikers when it is bound to the link source and it receives a rename notification through the **IAdviseSink::OnRename** method. A container application can also use the **IOleLink::SetSourceDisplayName** method to change a link's moniker.

The linked object's implementation of **IPersistStorage::Save** saves both the relative and the absolute moniker.

See Also

IOleLink::GetSourceMoniker, **IOleLink::SetSourceDisplayName**, **IOleUILinkContainer**, **OleUIEditLinks**

IOleLink::SetUpdateOptions Quick Info

Specifies how often a linked object should update its cached data.

```
HRESULT SetUpdateOptions(
  DWORD dwUpdateOpt      //Update option
);
```

Parameter

dwUpdateOpt

[in] Specifies how often a linked object should update its cached data. The legal values for *dwUpdateOpt* are taken from the enumeration **OLEUPDATE**.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The update option was successfully set.

Remarks

Notes to Callers

Your container application should call **IOleLink::SetUpdateOptions** when the end user changes the update option for a linked object.

The end user selects the update option for a linked object using the Links dialog box. If you use the **OleUIEditLinks** function to display this dialog box, you must implement the **IOleUILinkContainer** interface. The dialog box calls your **IOleUILinkContainer::SetLinkUpdateOptions** method to specify the update option chosen by the end user. Your implementation of this method should call the **IOleLink::SetUpdateOptions** method to pass the selected option to the linked object.

Notes on Provided Implementation

The default update option is OLEUDPATE_ALWAYS. The linked object's implementation of **IPersistStorage::Save** saves the current update option.

If OLEUDPATE_ALWAYS is specified as the update option, the linked object updates the link's caches in the following situations:

- When the update option is changed from manual to automatic, if the link source is running.
- Whenever the linked object binds to the link source.
- Whenever the link source is running and the linked object's **IOleObject::Close**, **IPersistStorage::Save**, or **IAdviseSink::OnSave** implementations are called.

For both manual and automatic links, the linked object updates the cache whenever the container application calls **IOleObject::Update** or **IOleLink::Update**.

See Also

IOleObject::Update, **IOleLink::GetUpdateOptions**, **IOleLink::Update**, **IOleUILinkContainer**, **OleUIEditLinks**

IOleLink::UnbindSource Quick Info

Deactivates the connection between a linked object and its link source.

HRESULT UnbindSource(*void***);**

Return Value

S_OK

The connection was deactivated.

Remarks

You typically do not call **IOleLink::UnbindSource** directly. When it's necessary to deactivate the connection to the link source, your container typically calls **IOleObject::Close** or **IUnknown::Release**; the linked object's implementation of these methods calls **IOleLink::UnbindSource**. The linked object's **IAdviseSink::OnClose** implementation also calls **IOleLink::UnbindSource**.

Notes on Provided Implementation

The linked object's implementation of **IOleLink::UnbindSource** does nothing if the link source is not currently bound. If the link source is bound, **IOleLink::UnbindSource** calls the link source's **IOleObject::Unadvise** and **IDataObject::DUnadvise** implementations to delete the advisory connections to the link source. The **IOleLink::UnbindSource** method also calls the compound document's **IOleClientSite::LockContainer** implementation to unlock the containing compound document. This undoes the lock on the container and the advisory connections that were established in **IOleLink::BindToSource**. **IOleLink::UnbindSource** releases all the linked object's interface pointers to the link source.

See Also

IAdviseSink::OnClose, **IDataObject::DUnadvise**, **IOleObject::Close**, **IOleObject::Unadvise**, **IOleLink::BindToSource**

IOleLink::Update Quick Info

Updates the compound document's cached data for a linked object. This involves binding to the link source, if it is not already bound.

```
HRESULT Update(
  IBindCtx *pbc      //Pointer to bind context to be used
);
```

Parameter

pbc

[in] Pointer to the **IBindCtx** interface on the bind context to be used in binding the link source. This parameter can be NULL. The bind context caches objects bound during the binding process, contains parameters that apply to all operations using the bind context, and provides the means by which the binding implementation should retrieve information about its environment. For more information, see **IBindCtx**.

Return Values

S_OK

All caches were updated successfully.

CACHE_E_NOCACHE_UPDATED

The bind operation worked but no caches were updated.

CACHE_S_SOMECACHES_NOTUPDATED

The bind operation worked but not all caches were updated.

OLE_E_CANT_BINDTOSOURCE

Unable to bind to the link source.

Remarks

Notes to Callers

Your container application should call **IOleLink::Update** if the end user updates the cached data for a linked object.

The end user can update the cached data for a linked object by choosing the Update Now button in the Links dialog box. If you use the **OleUIEditLinks** function to display the Links dialog box, you must implement the **IOleUILinkContainer** interface. The dialog box calls your implementations of **IOleUILinkContainer::UpdateLink** when the end user chooses the Update Now button. Your implementation of that method can call **IOleLink::Update**.

Your container application can also call **IOleObject::Update** to update a linked object, because that method – when called on a linked object – calls **IOleLink::Update**.

This method updates both automatic links and manual links. For manual links, calling **IOleLink::Update** or **IOleObject::Update** is the only way to update the caches. For more information on automatic and manual links, see **IOleLink::SetUpdateOptions**.

Notes on Provided Implementation

If *pbc* is non-NULL, the linked object's implementation of **IOleLink::Update** calls **IBindCtx::RegisterObjectBound** to register the bound link source. This ensures that the link source remains running until the bind context is released.

The current caches are left intact if the link source cannot be bound.

See Also

IBindCtx::RegisterObjectBound, **IOleLink::SetUpdateOptions**, **IOleObject::Update**, **IOleUILinkContainer**, **OleUIEditLinks**

IOleObject Quick Info

The **IOleObject** interface is the principal means by which an embedded object provides basic functionality to, and communicates with, its container.

When to Implement

An object application must implement this interface, along with at least **IDataObject** and **IPersistStorage**, for each type of embedded object that it supports. Although this interface contains 21 methods, only three are nontrivial to implement and must be fully implemented: **DoVerb**, **SetHostNames**, **and Close**. Six of the methods provide optional functionality, which, if not desired, can be implemented to return E_NOTIMPL: **SetExtent**, **InitFromData**, **GetClipboardData**, **SetColorScheme**, **SetMoniker**, and **GetMoniker**. The latter two methods are useful mainly for enabling links to embedded objects.

When to Use

Call the methods of this interface to enable a container to communicate with an embedded object. A container must call **DoVerb** to activate an embedded object, **SetHostNames** to communicate the names of the container application and container document, and Close to move an object from a running to a loaded state. Calls to all other methods are optional.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IOleObject Methods	Description
SetClientSite	Informs object of its client site in container.
GetClientSite	Retrieves object's client site.
SetHostNames	Communicates names of container application and container document.
Close	Moves object from running to loaded state.
SetMoniker	Informs object of its moniker.
GetMoniker	Retrieves object's moniker.
InitFromData	Initializes embedded object from selected data.
GetClipboardData	Retrieves a data transfer object from the Clipboard.
DoVerb	Invokes object to perform one of its enumerated actions ("verbs").
EnumVerbs	Enumerates actions ("verbs") for an object.
Update	Updates an object.
IsUpToDate	Checks if object is up to date.
GetUserClassID	Returns an object's class identifier.
GetUserType	Retrieves object's user-type name.

SetExtent	Sets extent of object's display area.
GetExtent	Retrieves extent of object's display area.
Advise	Establishes advisory connection with object.
Unadvise	Destroys advisory connection with object.
EnumAdvise	Enumerates object's advisory connections.
GetMiscStatus	Retrieves status of object.
SetColorScheme	Recommends color scheme to object application.

IOleObject::Advise

Quick Info

Establishes an advisory connection between a compound document object and the calling object's advise sink, through which the calling object receives notification when the compound document object is renamed, saved, or closed.

```
HRESULT Advise(
  IAdviseSink *pAdvSink,      //Pointer to advisory sink
  DWORD *pdwConnection        //Pointer to a token
);
```

Parameters

pAdvSink

[in] Pointer to the **IAdviseSink** interface on the advise sink of the calling object.

pdwConnection

[out] Pointer to a DWORD token that can be passed to **IOleObject::Unadvise** to delete the advisory connection.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

Advisory connection is successfully established.

Remarks

The **Advise** method sets up an advisory connection between an object and its container, through which the object informs the container's advise sink of close, save, rename, and link-source change events in the object. A container calls this method, normally as part of initializing an object, to register its advisory sink with the object. In return, the object sends the container compound-document notifications by calling **IAdviseSink** or **IAdviseSink2**.

If container and object successfully establish an advisory connection, the object receiving the call returns a nonzero value through *pdwConnection* to the container. If the attempt to establish an advisory connection fails, the object returns zero. To delete an advisory connection, the container calls **IOleObject::Unadvise** and passes this nonzero token back to the object.

An object can delegate the job of managing and tracking advisory events to an OLE advise holder, to which you obtain a pointer by calling **CreateOleAdviseHolder**. The returned **IOleAdviseHolder** interface has three methods for sending advisory notifications, as well as **Advise**, **Unadvise**, and **EnumAdvise** methods that are identical to those for **IOleObject**. Calls to **IOleObject:Advise**, **Unadvise**, or **EnumAdvise** are delegated to corresponding methods in the advise holder.

To destroy the advise holder, simply call Release on the **IOleAdviseHolder** interface.

See Also

CreateOleAdviseHolder, **IOleObject::UnAdvise**, **IOleObject::EnumAdvise**, **IOleAdviseHolder::Advise**

IOleObject::Close Quick Info

Changes an embedded object from the running to the loaded state. Disconnects a linked object from its link source.

```
HRESULT Close(
  DWORD dwSaveOption      //Indicates whether to save object before closing
);
```

Parameter

dwSaveOption

[in] DWORD that indicates whether the object is to be saved as part of the transition to the loaded state. Valid values are taken from the enumeration **OLECLOSE**.

Note The OLE 2 user model recommends that object applications do not prompt users before saving linked or embedded objects, including those activated in place. This policy represents a change from the OLE 1 user model, in which object applications always prompt the user to decide whether to save changes.

Return Values

S_OK

The object closed successfully.

OLE_E_PROMPTSAVECANCELLED

The user was prompted to save but chose the Cancel button from the prompt message box.

Remarks

Notes to Callers

A container application calls **IOleObject::Close** when it wants to move the object from a running to a loaded state. Following such a call, the object still appears in its container but is not open for editing. Calling **IOleObject::Close** on an object that is loaded but not running has no effect. Closing a linked object simply means disconnecting it.

Notes to Implementers

Upon receiving a call to **IOleObject::Close**, a running object should do the following:

- If the object has been changed since it was last opened for editing, it should request to be saved, or not, according to instructions specified in *dwSaveOption*. If the option is to save the object, then it should call its container's **IOleClientSite::SaveObject** interface.
- If the object has **IDataObject::DAdvise** connections with ADVF_DATAONSTOP flags, then it should send an OnDataChange notification. See **IDataObject::DAdvise** for details.
- If the object currently owns the Clipboard, it should empty it by calling **OleFlushClipboard**.
- If the object is currently visible, notify its container by calling **IOleClientSite::OnShowWindow** with the *fshow* argument set to FALSE.

- Send **IAdvise::OnClose** notifications to appropriate advise sinks.
- Finally, forcibly cut off all remoting clients by calling **CoDisconnectObject**.

If the object application is a local server (an EXE rather than a DLL), closing the object should also shut down the object application unless the latter is supporting other running objects or has another reason to remain in the running state. Such reasons might include the presence of **IClassFactory::LockServer** locks, end-user control of the application, or the existence of other open documents requiring access to the application.

Calling **IOleObject::Close** on a linked object disconnects it from, but does not shut down, its source application. A source application that is visible to the user when the object is closed remains visible and running after the disconnection and does not send an **OnClose** notification to the link container.

See Also

CoDisconnectObject, **IAdviseSink::OnClose**, **IClassFactory::LockServer**, **IDataObject::DAdvise**, **IOleClientSite::OnShowWindow**, **IOleClientSite::SaveObject**, **OLECLOSE**, **OleFlushClipboard**

IOleObject::DoVerb Quick Info

Requests an object to perform an action in response to an end-user's action. The possible actions are enumerated for the object in **IOleObject::EnumVerbs**.

```
HRESULT DoVerb(
  LONG iVerb,                    //Value representing verb to be performed
  LPMSG lpmsg,                   //Pointer to structure that describes Windows message
  IOleClientSite *pActiveSite,   //Pointer to active client site
  LONG lindex,                   //Reserved
  HWND hwndParent,               //Handle of window containing the object
  LPCRECT lprcPosRect            //Pointer to object's display rectangle
);
```

Parameters

iVerb

[in] Number assigned to the verb in the **OLEVERB** structure returned by **IOleObject::EnumVerbs**.

lpmsg

[in] Pointer to the MSG structure describing the event (such as a double-click) that invoked the verb. The caller should pass the MSG structure unmodified, without attempting to interpret or alter the values of any of the fields of *lpmsg*.

pActiveSite

[in] Pointer to the **IOleClientSite** interface on the object's active client site, where the event occurred that invoked the verb.

lindex

[in] Reserved for future use; must be zero.

hwndParent

[in] Handle of the document window containing the object. This and *lprcPosRect* together make it possible to open a temporary window for an object, where *hwndParent* is the parent window in which the object's window is to be displayed, and *lprcPosRect* defines the area available for displaying the object window within that parent. A temporary window is useful, for example, to a multimedia object that opens itself for playback but not for editing.

lprcPosRect

[in] Pointer to the RECT structure containing the coordinates, in pixels, that define an object's bounding rectangle in *hwndParent*. This and *hwndParent* together enable opening multimedia objects for playback but not for editing.

Return Values

S_OK

Object successfully invoked specified verb.

OLE_E_NOT_INPLACEACTIVE

iVerb set to OLEIVERB_UIACTIVATE or OLEIVERB_INPLACEACTIVATE and object is not already

visible.

OLE_E_CANT_BINDTOSOURCE

The object handler or link object cannot connect to the link source.

DV_E_LINDEX

Invalid *lindex*.

OLEOBJ_S_CANNOT_DOVERB_NOW

The verb is valid, but in the object's current state it cannot carry out the corresponding action.

OLEOBJ_S_INVALIDHWND

DoVerb was successful but *hwndParent* is invalid.

OLEOBJ_E_NOVERBS

The object does not support any verbs.

OLEOBJ_S_INVALIDVERB

Object does not recognize a positive verb number. Verb is treated as OLEIVERB_PRIMARY.

MK_E_CONNECT

Link source is across a network that is not connected to a drive on this machine.

OLE_E_CLASSDIFF

Class for source of link has undergone a conversion.

E_NOTIMPL

Object does not support in-place activation or does not recognize a negative verb number.

Remarks

A "verb" is an action that an OLE object takes in response to a message from its container. An object's container, or a client linked to the object, normally calls **IOleObject::DoVerb** in response to some end-user action, such as double-clicking on the object. The various actions that are available for a given object are enumerated in an **OLEVERB** structure, which the container obtains by calling **IOleObject::EnumVerbs**. **IOleObject::DoVerb** matches the value of *iVerb* against the *iVerb* member of the structure to determine which verb to invoke.

Through **IOleObject::EnumVerbs**, an object, rather than its container, determines which verbs (i.e., actions) it supports. OLE 2 defines seven verbs that are available, but not necessarily useful, to all objects. In addition, each object can define additional verbs that are unique to it. The following table describes the verbs defined by OLE:

Verb	Description
OLEIVERB_PRIMARY (0L)	Specifies the action that occurs when an end user double-clicks the object in its container. The object, not the container, determines this action. If the object supports in-place activation, the primary verb usually activates the object in place.
OLEIVERB_SHOW (-1)	Instructs an object to show itself for editing or viewing. Called to display newly inserted objects for initial

	editing and to show link sources. Usually an alias for some other object-defined verb.
OLEIVERB_OPEN (-2)	Instructs an object, including one that otherwise supports in-place activation, to open itself for editing in a window separate from that of its container. If the object does not support in-place activation, this verb has the same semantics as OLEIVERB_SHOW.
OLEIVERB_HIDE (-3)	Causes an object to remove its user interface from the view. Applies only to objects that are activated in-place.
OLEIVERB_UIACTIVATE (-4)	Activates an object in place, along with its full set of user-interface tools, including menus, toolbars, and its name in the title bar of the container window. If the object does not support in-place activation, it should return E_NOTIMPL.
OLEIVERB_INPLACEACTIVATE (-5)	Activates an object in place without displaying tools, such as menus and toolbars, that end users need to change the behavior or appearance of the object. Single-clicking such an object causes it to negotiate the display of its user-interface tools with its container. If the container refuses, the object remains active but without its tools displayed.
OLEIVERB_DISCARDUNDOSTATE (-6)	Used to tell objects to discard any undo state that they may be maintaining without deactivating the object.

Note to Callers

Containers call **IOleObject::DoVerb** as part of initializing a newly created object. Before making the call, containers should first call **IOleObject::SetClientSite** to inform the object of its display location and **IOleObject::SetHostNames** to alert the object that it is an embedded object and to trigger appropriate changes to the user interface of the object application in preparation for opening an editing window.

Like the **OleActivate** function in OLE 1, **IOleObject::DoVerb** automatically runs the OLE server application. If an error occurs during verb execution, the object application is shut down.

If an end user invokes a verb by some means other than selecting a command from a menu (say, by double-clicking or, more rarely, single-clicking an object), the object's container should pass a pointer (*lpmsg*) to a Windows **MSG** structure containing the appropriate message. For example, if the end user invokes a verb by double-clicking the object, the container should pass a **MSG** structure containing WM_LBUTTONDBLCLK, WM_MBUTTONDBLCLK, or WM_RBUTTONDBLCLK. If the container passes no message, *lpmsg* should be set to NULL. The object should ignore the *hwnd* member of the passed

MSG structure, but can use all the other **MSG** members.

If the object's embedding container calls **IOleObject::DoVerb**, the client-site pointer (*pClientSite*) passed to **DoVerb** is the same as that of the embedding site. If the embedded object is a link source, the pointer passed to **DoVerb** is that of the linking client's client site.

When **IOleObject::DoVerb** is invoked on an OLE link, it may return OLE_E_CLASSDIFF or MK_CONNECTMANUALLY. The link object returns the former error when the link source has been subjected to some sort of conversion while the link was passive. The link object returns the latter error when the link source is located on a network drive that is not currently connected to the caller's computer. The only way to connect a link under these conditions is to first call **QueryInterface**, ask for **IOleLink**, allocate a bind context, and run the link source by calling **IOleLink::BindToSource**.

Container applications that do not support general in-place activation can still use the *hwndParent* and *lprcPosRect* parameters to support in-place playback of multimedia files. Containers must pass valid *hwndParent* and *lprcPosRect* parameters to **IOleObject::DoVerb**.

Some code samples pass a *lindex* value of -1 instead of zero. The value -1 works but should be avoided in favor of zero. The *lindex* parameter is a reserved parameter, and for reasons of consistency Microsoft recommends assigning a zero value to all reserved parameters.

Notes to Implementers

In addition to the above verbs, an object can define in its **OLEVERB** structure additional verbs that are specific to itself. Positive numbers designate these object-specific verbs. An object should treat any unknown *positive* verb number as if it were the primary verb and return OLE_S_INVALIDVERB to the calling function. The object should ignore verbs with negative numbers that it does not recognize and return E_NOTIMPL.

If the verb being executed places the object in the running state, you should register the object in the Running Object Table (ROT) even if its server application doesn't support linking. Registration is important because the object at some point may serve as the source of a link in a container that supports links to embeddings. Registering the object with the ROT enables the link client to get a pointer to the object directly, instead of having to go through the object's container. To perform the registration, call **IOleClientSite::GetMoniker** to get the full moniker of the object, call the **GetRunningObjectTable** function to get a pointer to the ROT, and then call **IRunningObjectTable::Register**.

> **Note** When the object leaves the running state, remember to revoke the object's registration with the ROT by calling **IOleObject::Close**. If the object's container document is renamed while the object is running, you should revoke the object's registration and re-register it with the ROT, using its new name. The container should inform the object of its new moniker either by calling **IOleObject::SetMoniker** or by responding to the object's calling **IOleClientSite::GetMoniker**.

When showing a window as a result of **DoVerb**, it is very important for the object to explicitly call **SetForegroundWindow** on its editing window. This ensures that the object's window will be visible to the user even if another process originally obscured it. For more information about **SetForegroundWindow** and **SetActiveWindow**, see the *Win32 SDK*.

See Also

GetRunningObjectTable, **IOleClientSite::GetMoniker**, **IOleLink::BindToSource**, **IOleObject::Close**, **IOleObject::EnumVerbs**, **IOleObject::GetMoniker**, **IOleObject::SetMoniker**, **IRunningObjectTable::Register**, **OleRun**, **SetForegroundWindow**, **SetActiveWindow** in Win32

IOleObject::EnumAdvise Quick Info

Retrieves a pointer to an enumerator that can be used to enumerate the advisory connections registered for an object, so a container can know what to release prior to closing down.

```
HRESULT EnumAdvise(
  IEnumSTATDATA **ppenumAdvise      //Indirect pointer to enumerator object
);
```

Parameter

ppenumAdvise

[out] When there are existing advisory connections, indirect pointer to the location of the **IEnumSTATDATA** interface on the enumerator object. If the object does not have any advisory connections, the value will be NULL. NULL will also be the value if the method returns an error. Each time an object receives a successful call to **EnumAdvise**, it must increase the reference count on the pointer it returns. It is the caller's responsibility to call **Release** when it is done with the pointer.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

Enumerator returned successfully.

E_NOTIMPL

EnumAdvise is not implemented.

Remarks

The **EnumAdvise** method supplies an enumerator that provides a way for containers to keep track of advisory connections registered for their objects. A container normally would call this function so that it can instruct an object to release each of its advisory connections prior to closing down.

The enumerator to which you get access through **IOleObject::EnumAdvise** enumerates items of type **STATDATA**. Upon receiving the pointer, the container can then loop through **STATDATA** and call **IOleObject::Unadvise** for each enumerated connection.

The usual way to implement this function is to delegate the call to the **IOleAdviseHolder** interface. Only the *pAdvise* and *dwConnection* members of **STATDATA** are relevant for **IOleObject::EnumAdvise**.

See Also

IOleObject::Advise, **IOleObject::UnAdvise**

IOleObject::EnumVerbs Quick Info

Exposes a pull-down menu listing the verbs available for an object in ascending order by verb number.

```
HRESULT EnumVerbs(
  IEnumOleVerb **ppEnumOleVerb      //Indirect pointer to storage of enumerator object
);
```

Parameter

ppEnumOleVerb

[out] When successful, indirect pointer to where the **IEnumOLEVERB** interface on the new enumerator should be returned. Each time an object receives a call to **EnumVerbs**, it must increase the reference count on the pointer the method returns. It is the caller's responsibility to call **Release** when it is done with the pointer. If an error is returned, this parameter must be set to NULL.

Return Values

S_OK

Verb(s) enumerated successfully.

OLE_S_USEREG

Delegate to the default handler to use the entries in the registry to provide the enumeration.

OLEOBJ_E_NOVERBS

Object does not support any verbs.

Remarks

Notes to Callers

Containers call this method to expose a pull-down menu of the verbs available for their embedded objects. You may want your container to call **IOleObject::EnumVerbs** each and every time such a menu is selected in order to enable such objects as media clips, whose verbs may change while they are running, to update their menus. The default verb for a media clip, for example, changes from "Play" before it is activated to "Stop" once it is running.

Notes to Implementers

The default handler's implementation of **IOleObject::EnumVerbs** uses the registry to enumerate an object's verbs. If an object application is to use the default handler's implementation, it should return OLE_S_USEREG.

The enumeration returned is of type **IEnumOLEVERB**:

```
typedef Enum < OLEVERB > IEnumOLEVERB;
```

where **OLEVERB** is defined as:

```
typedef struct tagOLEVERB
{
    LONG    iVerb;
```

```
    LPOLESTR   lpszVerbName;
    DWORD   fuFlags;
    DWORD   grfAttribs;
} OLEVERB;
```

The following table describes the members of the **OLEVERB** structure:

OLEVERB Member	Description
iVerb	Verb number being enumerated. If the object supports OLEIVERB_OPEN, OLEIVERB_SHOW and/or OLEIVERB_HIDE (or another predefined verb), these will be the first verbs enumerated, since they have the lowest verb numbers.
lpszVerbName	Name of the verb. In Windows, this value, along with optional embedded ampersand characters to indicate accelerator keys, can be passed to the **AppendMenu** function. On the Macintosh, the following metacharacters may be passed along with this value: • ! marks the menu item with the subsequent character • < sets the character style of the item • (disables the item. The metacharacters / and ^ are not permitted.
fuFlags	In Windows, a group of flags taken from the flag constants beginning with MF_ defined in **AppendMenu**. Containers should use these flags in building an object's verb menu. All Flags defined in **AppendMenu** are supported except for: • MF_BITMAP • MF_OWNERDRAW • MF_POPUP
grfAttribs	In Windows, a group of flag bits taken from the enumeration **OLEVERBATTRIB**. The flag OLEVERBATTRIB_NEVERDIRTIES indicates that executing this verb will not cause the object to become dirty and is therefore in need of saving to persistent storage. OLEVERBATTRIB_ONCONTAINERMENU indicates that this verb should be placed on the container's menu of object verbs when the object is selected. OLEIVERB_HIDE, OLEIVERB_SHOW, and OLEIVERB_OPEN never have this value set.

For more information on the Windows **AppendMenu** function, see the Microsoft *Win32 SDK*.

See Also

IOleObject::DoVerb, OleRegEnumVerbs

IOleObject::GetClientSite Quick Info

Obtains a pointer to an embedded object's client site.

```
HRESULT GetClientSite(
  IOleClientSite **ppClientSite      //Indirect pointer to storage of object's client site
);
```

Parameter

ppClientSite

[out] Indirect pointer to an **IOleClientSite** interface on the object's client site. If an object does not yet know its client site, or an error has occurred, this parameter must be set to NULL. Each time an object receives a call to **GetClientSite**, it must increase the reference count on the pointer the method returns. It is the caller's responsibility to call **Release** when it is done with the pointer.

Return Value

S_OK

Client site pointer returned successfully.

Remarks

Link clients most commonly call the **IOleObject::GetClientSite** method in conjunction with the **IOleClientSite::GetContainer** method to traverse a hierarchy of nested objects. A link client calls **IOleObject::GetClientSite** to get a pointer to the link source's client site. The client then calls **IOleClientSite::GetContainer** to get a pointer to the link source's container. Finally, the client calls **IOleContainer::QueryInterface** to get **IOleObject** and **IOleObject::GetClientSite** to get the container's client site within its container. By repeating this sequence of calls, the caller can eventually retrieve a pointer to the master container in which all the other objects are nested.

Notes to Callers

The returned client-site pointer will be NULL if an embedded object has not yet been informed of its client site. This will be the case with a newly loaded or created object when a container has passed a NULL client-site pointer to one of the object-creation helper functions but has not yet called **IOleObject::SetClientSite** as part of initializing the object.

See Also

IOleObject::SetClientSite

IOleObject::GetClipboardData Quick Info

Retrieves a data object containing the current contents of the embedded object on which this method is called. Using the pointer to this data object, it is possible to create a new embedded object with the same data as the original.

```
HRESULT GetClipboardData(
  DWORD dwReserved,          //Reserved
  IDataObject **ppDataObject //Indirect pointer to storage of data object
);
```

Parameters

dwReserved

[in] Reserved for future use; must be zero.

ppDataObject

[out] Indirect pointer to the **IDataObject** interface on the data object. If an error is returned, this parameter must be set to NULL. Each time an object receives a call to **GetClipboardData**, it must increase the reference count on the pointer that the method returns. It is the caller's responsibility to call **Release** when it is done with the pointer.

Return Values

S_OK

The data transfer object is successfully returned.

E_NOTIMPL

GetClipboardData is not supported.

OLE_E_NOTRUNNING

The object is not running.

Remarks

You can use the **GetClipboardData** method to convert a linked object to an embedded object, in which case the container application would call **IOleObject::GetClipboardData** and then pass the data received to **OleCreateFromData**. This method returns a pointer to a data object that is identical to what would have been passed to the Clipboard by a standard copy operation.

Notes to Callers

If you want a stable snapshot of the current contents of an embedded object, call **IOleObject::GetClipboardData**. Should the data change, you will need to call the function again for an updated snapshot. If you want the caller to be informed of changes that occur to the data, call **IDataObject::QueryInterface**, then call **IDataObject::DAdvise**.

Notes to Implementers

If you implement this function, you must return an **IDataObject** pointer for an object whose data will not change.

See Also

IDataObject, IOleObject::InitFromData, IUnknown::QueryInterface, OleCreateFromData

IOleObject::GetExtent Quick Info

Retrieves a running object's current display size.

```
HRESULT GetExtent(
  DWORD dwDrawAspect,     //Value indicating object aspect
  SIZEL *psizel           //Pointer to storage of object size limit
);
```

Parameters

dwDrawAspect

[in] Value indicating the aspect of the object whose limit is to be retrieved; the value is obtained from the enumerations **DVASPECT** and from **DVASPECT2**. Note that newer objects and containers that support optimized drawing interfaces support the **DVASPECT2** enumeration values. Older objects and containers that do not support optimized drawing interfaces may not support **DVASPECT2**. The most common value for this method is DVASPECT_CONTENT, which specifies a full rendering of the object within its container.

psizel

[out] Pointer to where the object's size is to be returned.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Extent information successfully returned.

Remarks

A container calls **IOleObject::GetExtent** on a running object to retrieve its current display size. If the container can accommodate that size, it will normally do so because the object, after all, knows what size it should be better than the container does. A container normally makes this call as part of initializing an object.

The display size returned by **IOleObject::GetExtent** may differ from the size last set by **IOleObject::SetExtent** because the latter method dictates the object's display space at the time the method is called but does not necessarily change the object's native size, as determined by its application.

Note This method must return the same size as DVASPECT_CONTENT for all the new aspects in **DVASPECT2**. **IViewObject2::GetExtent** must do the same thing.

If one of the new aspects is requested in *dwAspect*, this method can either fail or return the same rectangle as for the DVASPECT_CONTENT aspect.

Notes to Callers

Because a container can make this call only to a running object, the container must instead call

IViewObject2::GetExtent if it wants to get the display size of a loaded object from its cache.

Notes to Implementers

Implementation consists of filling the *sizel* structure with an object's height and width.

See Also

DVASPECT, **DVASPECT2**, **IOleObject::SetExtent**, **IViewObject2::GetExtent**

IOleObject::GetMiscStatus Quick Info

Returns a value indicating the status of an object at creation and loading.

```
HRESULT GetMiscStatus(
    DWORD dwAspect,      //Value indicating object aspect
    DWORD *pdwStatus     //Pointer to storage of status information
);
```

Parameters

dwAspect

[in] Value indicating the aspect of an object about which status information is being requested. The value is obtained from the enumeration **DVASPECT** (see "FORMATETC Data Structure").

pdwStatus

[out] Pointer to where the status information is returned. May not be NULL.

Return Values

S_OK

Status information returned successfully.

OLE_S_USEREG

Delegate the retrieval of miscellaneous status information to the default handler's implementation of this method.

CO_E_CLASSNOTREG

There is no CLSID registered for the object.

CO_E_READREGDB

Error accessing the registry.

Remarks

A container normally calls **IOleObject::GetMiscStatus** when it creates or loads an object in order to determine how to display the object and what types of behaviors it supports.

Objects store status information in the registry. If the object is not running, the default handler's implementation of **IOleObject::GetMiscStatus** retrieves this information from the registry. If the object is running, the default handler invokes **IOleObject::GetMiscStatus** on the object itself.

The information that is actually stored in the registry varies with individual objects. The status values to be returned are defined in the enumeration **OLEMISC**.

Notes to Implementers

Implementation normally consists of delegating the call to the default handler.

See Also

DVASPECT, **FORMATETC**, **OLEMISC**

IOleObject::GetMoniker Quick Info

Retrieves an embedded object's moniker, which the caller can use to link to the object.

```
HRESULT GetMoniker(
  DWORD dwAssign,          //Specifies how moniker is assigned to object
  DWORD dwWhichMoniker,    //Specifies which moniker is assigned
  IMoniker **ppmk          //Indirect pointer to location of object's moniker
);
```

Parameters

dwAssign

[in] Determines how the moniker is assigned to the object. Depending on the value of *dwAssign*, **IOleObject::GetMoniker** does one of the following:

- Obtains a moniker only if one has already been assigned,
- Forces assignment of a moniker, if necessary, in order to satisfy the call, or
- Obtains a temporary moniker.

Values for *dwAssign* are specified in the enumeration **OLEGETMONIKER**.

Note You cannot pass OLEGETMONIKER_UNASSIGN when calling **IOleObject::GetMoniker**. This value is valid only when calling **IOleClientSite::GetMoniker**.

dwWhichMoniker

[in] Specifies the form of the moniker being requested. Valid values are taken from the enumeration **OLEWHICHMK**.

ppmk

[out] Indirect pointer to the location of the **IMoniker**interface on the object's moniker. If an error is returned, this parameter must be set to NULL. Each time an object receives a call to **GetMoniker**, it must increase the reference count on the pointer the method returns. It is the caller's responsibility to call Release when it is done with the pointer.

Remarks

The **IOleObject::GetMoniker** method returns an object's moniker. Like **IOleObject::SetMoniker**, this method is important only in the context of managing links to embedded objects and even in that case is optional. A potential link client that requires an object's moniker to bind to the object can call this method to obtain that moniker. The default implementation of **IOleObject::GetMoniker** calls the **IOleClientSite::GetMoniker**, returning E_UNEXPECTED if the object is not running or does not have a valid pointer to a client site.

See Also

CreateItemMoniker, **IOleClientSite::GetMoniker**, **IOleObject::SetMoniker**, **OLEGETMONIKER**, **OLEWHICHMK**

IOleObject::GetUserClassID Quick Info

Returns an object's class identifier, the CLSID corresponding to the string identifying the object to an end user.

```
HRESULT GetUserClassID(
  CLSID *pClsid       //Pointer to the class identifier
);
```

Parameter

pClsid

[out] Pointer to the class identifier (CLSID) to be returned. An object's CLSID is the binary equivalent of the user-type name returned by **IOleObject::GetUserType**.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

CLSID returned successfully.

Remarks

GetUserClassID returns the CLSID associated with the object in the registration database. Normally, this value is identical to the CLSID stored with the object, which is returned by **IPersist::GetClassID**. For linked objects, this is the CLSID of the last bound link source. If the object is running in an application different from the one in which it was created and for the purpose of being edited is emulating a class that the container application recognizes, the CLSID returned will be that of the class being emulated rather than that of the object's own class.

See Also

IOleObject::GetUserType, **IPersist::GetClassID**, **OleDoAutoConvert**, **OleGetAutoConvert**, **OleSetAutoConvert**, **GetConvertStg**, **SetConvertStg**

IOleObject::GetUserType Quick Info

Retrieves the user-type name of an object for display in user-interface elements such as menus, list boxes, and dialog boxes.

```
HRESULT GetUserType(
  DWORD dwFormOfType,      //Specifies form of type name
  LPOLESTR *pszUserType    //Indirect pointer to storage of string
);
```

Parameters

dwFormOfType

[in] Value specifying the form of the user-type name to be presented to users. Valid values are obtained from the **USERCLASSTYPE** enumeration.

pszUserType

[out] Indirect pointer to where the user-type string will be placed. The caller must free *lpszUserType* using the current **IMalloc** instance. If an error is returned, this parameter must be set to NULL.

Return Values

S_OK

The object's user-type name is successfully returned.

OLE_S_USEREG

Delegate to the default handler's implementation using the registry to provide the requested information.

Remarks

Containers call **IOleObject::GetUserType** in order to represent embedded objects in list boxes, menus, and dialog boxes by their normal, user-recognizable names. Examples include "Word Document," "Excel Chart," and "Paintbrush Object." The information returned by **IOleObject::GetUserType** is the user-readable equivalent of the binary class identifier returned by **IOleObject::GetUserClassID**.

Notes to Callers

The default handler's implementation of **IOleObject::GetUserType** uses the object's class identifier (the *pClsid* parameter returned by **IOleObject::GetUserClassID**) and the *dwFormOfType* parameter together as a key into the registry. If an entry is found that matches the key exactly, then the user type specified by that entry is returned. If only the CLSID part of the key matches, then the lowest-numbered entry available (usually the full name) is used. If the CLSID is not found, or there are no user types registered for the class, the user type currently found in the object's storage is used.

You should not cache the string returned from **GetUserType.** Instead, call this method each and every time the string is needed. This guarantees correct results when the embedded object is being converted from one type into another without the caller's knowledge. Calling this method is inexpensive because the default handler implements it using the registry.

Notes to Implementers

You can use the implementation provided by the default handler by returning OLE_S_USEREG as your application's implementation of this method. If the user type name is an empty string, the message "Unknown Object" is returned.

You can call the OLE helper function **OleRegGetUserType** to return the appropriate user type.

See Also

IOleObject::SetHostNames, **IOleObject::GetUserClassID**, **OleRegGetUserType**, **ReadFmtUserTypeStg**, **USERCLASSTYPE**

IOleObject::InitFromData Quick Info

Initializes a newly created object with data from a specified data object, which can reside either in the same container or on the Clipboard.

```
HRESULT InitFromData(
  IDataObject *pDataObject,     //Pointer to data object
  BOOL fCreation,               //Specifies how object is created
  DWORD dwReserved              //Reserved
);
```

Parameters

pDataObject

[in] Pointer to the **IDataObject** interface on the data object from which the initialization data is to be obtained. This parameter can be NULL, which indicates that the caller wants to know if it is worthwhile trying to send data; that is, whether the container is capable of initializing an object from data passed to it. The data object to be passed can be based on either the current selection within the container document or on data transferred to the container from an external source.

fCreation

[in] TRUE indicates the container is inserting a new object inside itself and initializing that object with data from the current selection; FALSE indicates a more general programmatic data transfer, most likely from a source other than the current selection.

dwReserved

[in] Reserved for future use; must be zero.

Return Values

S_OK

If *pDataObject* is not NULL, the object successfully attempted to initialize itself from the provided data; if *pDataObject* is NULL, the object is able to attempt a successful initialization.

S_FALSE

If *pDataObject* is not NULL, the object made no attempt to initialize itself; if *pDataObject* is NULL, the object cannot attempt to initialize itself from the data provided.

E_NOTIMPL

The object does not support **InitFromData**.

OLE_E_NOTRUNNING

The object is not running and therefore cannot perform the operation.

Remarks

This method enables a container document to insert within itself a new object whose content is based on a current data selection within the container. For example, a spreadsheet document may want to create a graph object based on data in a selected range of cells.

Using this method, a container can also replace the contents of an embedded object with data transferred

from another source. This provides a convenient way of updating an embedded object.

Notes to Callers

Following initialization, the container should call **IOleObject::GetMiscStatus** to check the value of the **OLEMISC_INSERTNOTREPLACE** bit. If the bit is on, the new object inserts itself following the selected data. If the bit is off, the new object replaces the selected data.

Notes to Implementers

A container specifies whether to base a new object on the current selection by passing either TRUE or FALSE to the *fCreation* parameter.

If *fCreation* is TRUE, the container is attempting to create a *new* instance of an object, initializing it with the selected data specified by the data object.

If *fCreation* is FALSE, the caller is attempting to replace the object's current contents with that pointed to by *pDataObject*. The usual constraints that apply to an object during a paste operation should be applied here. For example, if the type of the data provided is unacceptable, the object should fail to initialize and return S_FALSE.

If the object returns S_FALSE, it cannot initialize itself from the provided data.

See Also

IOleObject::GetMiscStatus, **IDataObject::SetData**

IOleObject::IsUpToDate Quick Info

Checks recursively whether or not an object is up to date.

HRESULT IsUpToDate();

Return Values

S_OK

Object is up to date.

S_FALSE

Object is not up to date.

OLE_E_UNAVAILABLE

Status of object cannot be determined in a timely manner.

Remarks

The **IsUpToDate** method provides a way for containers to check recursively whether or not all objects are up to date. That is, when the container calls this method on the first object, the object in turn calls it for all its own objects, and they in turn for all of theirs, until all objects have been checked.

Notes to Implementers

Because of the recursive nature of **IOleObject:IsUpToDate**, determining whether an object is out-of-date, particularly one containing one or more other objects, can be as time-consuming as simply updating the object in the first place. If you would rather avoid lengthy queries of this type, make sure that **IOleObject::IsUpToDate** returns OLE_E_UNAVAILABLE. In cases where the object to be queried is small and contains no objects itself, thereby making an efficient query possible, this method can return either S_OK or S_FALSE.

See Also

IOleObject::UpDate

IOleObject::SetClientSite Quick Info

Informs an embedded object of its display location, called a "client site," within its container.

```
HRESULT SetClientSite(
  IOleClientSite *pClientSite      //Pointer to an embedded object's client site
);
```

Parameter

pClientSite

[in] Pointer to the **IOleClientSite** interface on the container application's client-site.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

Client site successfully set.

Remarks

Within a compound document, each embedded object has its own client site – the place where it is displayed and through which it receives information about its storage, user interface, and other resources. **IOleObject::SetClientSite** is the only method enabling an embedded object to obtain a pointer to its client site.

Notes to Callers

A container can notify an object of its client site either at the time the object is created or, subsequently, when the object is initialized.

When creating or loading an object, a container may pass a client-site pointer (along with other arguments) to one of the following helper functions: **OleCreate**, **OleCreateFromFile**, **OleCreateFromData** or **OleLoad**. These helper functions load an object handler for the new object and call **IOleObject::SetClientSite** on the container's behalf before returning a pointer to the new object.

Passing a client-site pointer informs the object handler that the client site is ready to process requests. If the client site is unlikely to be ready immediately after the handler is loaded, you may want your container to pass a NULL client-site pointer to the helper function. The NULL pointer says that that no client site is available and thereby defers notifying the object handler of the client site until the object is initialized. In response, the helper function returns a pointer to the object, but upon receiving that pointer the container must call **SetClientSite** as part of initializing the new object.

Notes to Implementers

Implementation consists simply of incrementing the reference count on, and storing, the pointer to the client site.

See Also

IOleClientSite, **IOleObject::GetClientSite**, **OleCreate**, **OleCreateFromFile**, **OleCreateFromData**, **OleLoad**

IOleObject::SetColorScheme Quick Info

Specifies the color palette that the object application should use when it edits the specified object.

```
HRESULT SetColorScheme(
  LOGPALETTE *pLogpal       //Pointer to a structure
);
```

Parameter

pLogpal

[in] Pointer to a **LOGPALETTE** structure that specifies the recommended palette.

Return Values

S_OK

Color palette received successfully.

E_NOTIMPL

Object does not support setting palettes.

OLE_E_PALETTE

Invalid **LOGPALETTE** structure pointed to by *lpLogPal*.

OLE_E_NOTRUNNING

Object must be running to perform this operation.

Remarks

The **IOleObject::SetColorScheme** method sends the container application's recommended color palette to the object application, which is not obliged to use it.

Notes to Implementers

Upon receiving the palette, the object application should:

1. Allocate and fill in its own **LOGPALETTE** structure with the colors specified in the container application's **LOGPALETTE** structure.
2. Call **CreatePalette** to create a palette from the resulting **LOGPALETTE** structure. This palette can be used to render objects and color menus as the user edits objects in the document.

The first palette entry in the **LOGPALETTE** structure specifies the foreground color recommended by the container. The second palette entry specifies the recommended background color. The first half of the remaining palette entries are fill colors and the second half are colors for the lines and text.

Container applications typically specify an even number of palette entries. If a container specifies an odd number of entries, the object application should assume that the first half of the total entries plus one designate fill colors, while the remainder designate line and text colors. For example, if there are five entries, the first three should be interpreted as fill colors and the last two as line and text colors.

IOleObject::SetExtent Quick Info

Informs an object of how much display space its container has assigned it.

```
HRESULT SetExtent(
  DWORD dwDrawAspect,     //DVASPECT value
  SIZEL  *psizel          //Pointer to size limit for object
);
```

Parameters

dwDrawAspect

[in] DWORD that describes which form, or "aspect," of an object is to be displayed. The object's container obtains this value from the enumeration **DVASPECT** (refer to the **FORMATETC** enumeration). The most common aspect is DVASPECT_CONTENT, which specifies a full rendering of the object within its container. An object can also be rendered as an icon, a thumbnail version for display in a browsing tool, or a print version, which displays the object as it would be rendered using the File Print command.

psizel

[in] Pointer to the size limit for the object.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The object has resized successfully.

OLE_E_NOTRUNNING

The object is not running.

Remarks

A container calls **IOleObject::SetExtent** when it needs to dictate to an embedded object the size at which it will be displayed. Often, this call occurs in response to an end user resizing the object window. Upon receiving the call, the object, if possible, should recompose itself gracefully to fit the new window.

Whenever possible, a container seeks to display an object at its finest resolution, sometimes called the object's *native size*. All objects, however, have a default display size specified by their applications, and in the absence of other constraints, this is the size they will use to display themselves. Since an object knows its optimum display size better than does its container, the latter normally requests that size from a running object by calling **IOleObject::GetExtent**. Only in cases where the container cannot accommodate the value returned by the object does it override the object's preference by calling **IOleObject::SetExtent**.

Notes to Callers

You can call **SetExtent** on an object only when the object is running. If a container resizes an object while an object is not running, the container should keep track of the object's new size but defer calling **IOleObject::SetExtent** until a user activates the object. If the OLEMISC_RECOMPOSEONRESIZE bit is set on an object, its container should force the object to run before calling **OleObject::SetExtent**.

As noted above, a container may want to delegate responsibility for setting the size of an object's display site to the object itself, by calling **IOleObject::GetExtent**.

Notes to Implementers

You may want to implement this method so that your object rescales itself to match as closely as possible the maximum space available to it in its container.

If an object's size is fixed, that is, if it cannot be set by its container, **OleObject::SetExtent** should return E_FAIL. This is always the case with linked objects, whose sizes are set by their link sources, not by their containers.

See Also

IAdviseSink::OnViewChange, **IOleObject::GetExtent**, **IViewObject2::GetExtent**

IOleObject::SetHostNames Quick Info

Provides an object with the name of its container application and the compound document in which it is embedded.

```
HRESULT SetHostNames(
  LPCOLESTR szContainerApp,     //Pointer to name of container application
  LPCOLESTR szContainerObj      //Pointer to name of container document
);
```

Parameters

szContainerApp

[in] Pointer to the name of the container application in which the object is running.

szContainerObj

[in] Pointer to the name of the compound document that contains the object. If you do not wish to display the name of the compound document, you can set this parameter to NULL.

Return Value

S_OK

Window title information set successfully.

Remarks

When a container initializes an embedded object, it calls this function to inform the object of the names of both the container application and container document. When the object is opened for editing, it displays these names in the title bar of its window.

Notes to Callers

Call **SetHostNames** only for embedded objects, because for linked objects, the link source supplies its own separate editing window and title bar information.

Notes to Implementers

An object's application of **SetHostNames** should include whatever modifications to its user interface may be appropriate to an object's embedded state. Such modifications typically will include adding and removing menu commands and altering the text displayed in the title bar of the editing window.

The complete window title for an embedded object in an SDI container application or an MDI application with a maximized child window should appear as follows:

<*object application name*> - <*object short type*> in <*container document*>

Otherwise, the title should be:

<*object application name*> - <*container document*>

The "object short type" refers to a form of an object's name short enough to be displayed in full in a list box.

Since these identifying strings are not stored as part of the persistent state of the object, **IOleObject::SetHostNames** must be called each time the object loads or runs.

See Also

IOleObject::GetUserType

IOleObject::SetMoniker Quick Info

Notifies an object of its container's moniker, the object's own moniker relative to the container, or the object's full moniker.

```
HRESULT SetMoniker(
    DWORD dwWhichMoniker,     //Value specifying moniker being set
    IMoniker *pmk             //Pointer to moniker
);
```

Parameters

dwWhichMoniker

[in] Value specifying which moniker is passed in *pmk*. Values are from the enumeration **OLEWHICHMK**.

pmk

[in] Pointer to where to return the moniker.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

Moniker successfully set.

Remarks

A container that supports links to embedded objects must be able to inform an embedded object when its moniker has changed. Otherwise, subsequent attempts by link clients to bind to the object will fail. The **IOleObject::SetMoniker** method provides one way for a container to communicate this information.

The container can pass either its own moniker, an object's moniker relative to the container, or an object's full moniker. In practice, if a container passes anything other than an object's full moniker, each object calls the container back to request assignment of the full moniker, which the object requires to register itself in the running object table.

The moniker of an object relative to its container is stored by the object handler as part of the object's persistent state. The moniker of the object's container, however, must not be persistently stored inside the object because the container can be renamed at any time.

Notes to Callers

A container calls **IOleObject::SetMoniker** when the container has been renamed, and the container's embedded objects currently or can potentially serve as link sources. Containers call **SetMoniker** mainly in the context of linking because an embedded object is already aware of its moniker. Even in the context of linking, calling this method is optional because objects can call **IOleClientSite::GetMoniker** to force assignment of a new moniker.

Note to Implementers

Upon receiving a call to **SetMoniker**, an object should register its full moniker in the running object table and send OnRename notification to all advise sinks that exist for the object.

See Also

CreateItemMoniker, IAdviseSink::OnRename, IOleClientSite::GetMoniker, IOleObject::GetMoniker

IOleObject::Unadvise Quick Info

Deletes a previously established advisory connection.

```
HRESULT Unadvise(
  DWORD dwConnection      //Token
);
```

Parameter

dwConnection

[in] Contains a token of nonzero value, which was previously returned from **IOleObject::Advise** through its *pdwConnection* parameter.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

Advisory connection deleted successfully.

OLE_E_NOCONNECTION

dwConnection does not represent a valid advisory connection.

Remarks

Normally, containers call **IOleObject::Unadvise** at shutdown or when an object is deleted. In certain cases, containers can call this method on objects that are running but not currently visible as a way of reducing the overhead of maintaining multiple advisory connections. The easiest way to implement this method is to delegate the call to **IOleAdviseHolder::Unadvise**.

See Also

IOleObject::Advise, **IOleObject::EnumAdvise**, **IOleAdviseHolder::Unadvise**

IOleObject::Update Quick Info

Updates an object handler's or link object's data or view caches.

HRESULT Update();

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

All caches are up to date.

OLE_E_CANT_BINDTOSOURCE

Cannot run object to get updated data. The object is for some reason unavailable to the caller.

CACHE_E_NOCACHE_UPDATED

No caches were updated.

CACHE_S_SOMECACHES_NOTUPDATED

Some caches were not updated.

Remarks

The **Update** method provides a way for containers to keep data updated in their linked and embedded objects. A link object can become out-of-date if the link source has been updated. An embedded object that contains links to other objects can also become out of date. An embedded object that does not contain links cannot become out of date because its data is not linked to another source.

Notes to Implementers

When a container calls a link object's **IOleObject::Update** method, the link object finds the link source and gets a new presentation from it. This process may also involve running one or more object applications, which could be time-consuming.

When a container calls an embedded object's **IOleObject::Update** method, it is requesting the object to update all link objects it may contain. In response, the object handler recursively calls **IOleObject::Update** for each of its own linked objects, running each one as needed.

See Also

IOleObject::IsUpToDate

IOleParentUndoUnit Quick Info

The **IOleParentUndoUnit** interface enables undo units to contain child undo units. For example, a complex action can be presented to the end user as a single undo action even though a number of separate actions are involved. All the subordinate undo actions are contained within the top-level, parent undo unit.

A parent undo unit is initially created using the **IOleUndoManager::Open** method. Then, to add another parent unit nested within an existing parent unit, you call **IOleParentUndoUnit::Open**. While a parent unit is open, the undo manager adds undo units to it by calling **IOleParentUndoUnit::Add**. When the undo manager closes a top-level parent, the entire undo unit with its nested subordinates is placed on top of the undo stack.

This interface is derived from **IOleUndoUnit** and supports all the methods on that interface.

If an object needs to create a parent unit, there are several cases to consider:

- To create an enabling parent unit, the object calls **IOleUndoManager::GetOpenParentState** on the undo manager and checks the return value. If the value is S_FALSE, then the object creates the enabling parent and opens it. If the return value is S_OK, then there is a parent already open. If the open parent is blocked (UAS_BLOCKED bit set), or an enabling parent (UAS_BLOCKED and UAS_NOPARENTENABLE bits not set), then there is no need to create the enabling parent. If the currently open parent is a disabling parent (UAS_NOPARENTENABLE bit set), then the enabling parent should be created and opened to re-enable adding undo units. Note that UAS_NORMAL has a value of zero, which means it is the absence of all other bits and is not a bit flag that can be set. If comparing **pdwState* against UAS_NORMAL, mask out unused bits from **pdwState* with UAS_MASK to allow for future expansion.
- To create a blocked parent, the object calls **IOleUndoManager::GetOpenParentState** and checks for an open parent that is already blocked. If one exists, then there is no need to create the new blocking parent. Otherwise, the object creates it and opens it on the stack.
- To create a disabling parent, the object calls **IOleUndoManager::GetOpenParentState** and checks for an open parent that is blocked or disabling. If either one exists it is unnecessary to create the new parent. Otherwise, the object creates the parent and opens it on the stack.

In the event that both the UAS_NOPARENTENABLE and UAS_BLOCKED flags are set, the flag that is most relevant to the caller should be used with UAS_NOPARENTENABLE taking precedence. For example, if an object is creating a simple undo unit, it should pay attention to the UAS_NOPARENTENABLE flag and clear the undo stack. If it is creating an enabling parent unit, then it should pay attention to the UAS_BLOCKED flag and skip the creation.

When the parent undo unit is marked blocked, it discards any undo units it receives.

When to Implement

An undo unit that is capable of containing child undo units implements this interface in addition to **IOleUndoUnit**.

When to Use

The undo manager calls the methods in this interface to add child undo units.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.

AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleUndoUnit Methods	**Description**
Do	Instructs the undo unit to carry out its action.
GetDescription	Returns a string that describes the undo unit and can be used in the undo or redo user interface.
GetUnitType	Returns the CLSID and a type identifier for the undo unit.
OnNextAdd	Notifies the last undo unit in the collection that a new unit has been added.

IOleParentUndoUnit Methods	**Description**
Open	Opens a new parent undo unit, which becomes part of the containing unit's undo stack.
Close	Closes the most recently opened parent undo unit.
Add	Adds a simple undo unit to the collection.
FindUnit	Indicates if the specified unit is a child of this undo unit or one of its children, that is if the specified unit is part of the hierarchy in this parent unit.
GetParentState	Returns state information about the innermost open parent undo unit.

See Also

IOleUndoManager, **IOleUndoUnit**

IOleParentUndoUnit::Add Quick Info

Adds a simple undo unit to the collection.

```
HRESULT Add(
  IOleUndoUnit* pUU          //Pointer to undo unit to be added
);
```

Parameters

pUU

[in] Pointer to undo unit to be added.

Return Values

S_OK

The specified unit was successfully added or the parent unit was blocked.

Remarks

The parent undo unit or undo manager must accept any undo unit given to it, unless it is blocked. If it is blocked, it should do nothing but return S_OK.

See Also

IOleUndoManager::Add

IOleParentUndoUnit::Close Quick Info

Closes the specified parent undo unit.

```
HRESULT Close(
    IOleParentUndoUnit* pPUU,    //Pointer to the currently open parent unit
    BOOL fCommit                 //Indicates whether to keep or discard the unit
);
```

Parameters

pPUU

[in] Pointer to the currently open parent unit to be closed.

fCommit

[in] Indicates whether to keep or discard the unit. If TRUE, the unit is kept in the collection. If FALSE, the unit is discarded. This parameter is used to allow the client to discard an undo unit under construction if an error or a cancellation occurs.

Return Values

S_OK

The parent unit had open child units and it was successfully closed.

S_FALSE

The parent undo unit did not have an open child and it was successfully closed.

E_INVALIDARG

If *pPUU* does not match the currently open parent undo unit, then implementations of this method should return E_INVALIDARG without changing any internal state unless the parent unit is blocked.

Remarks

A parent undo unit knows it is being closed when it returns S_FALSE from this method. At that time, it should terminate any communication with other objects which may be giving data to it through private interfaces.

Note to Implementers

To process a **Close**, a parent undo unit first checks to see if it has an open child unit. If it does not, it returns S_FALSE.

If it does have a child unit open, it calls the **Close** method on the child. If the child returns S_FALSE, then the parent undo unit verifies that *pPUU* points to the child unit, and closes that child undo unit. If the child returns S_OK then it handled the **Close** internally and its parent should do nothing.

If the parent unit is blocked, it should check the *pPUU* parameter to determine the appropriate return code. If *pPUU* is pointing to itself, then it should return S_FALSE.

Otherwise, it should return S_OK; the *fCommit* parameter is ignored; and no action is taken.

If *pPUU* does not match the currently open parent undo unit, then implementations of this method should

return E_INVALIDARG without changing any internal state. The only exception to this is if the unit is blocked.

Note to Callers

An error return indicates a fatal error condition.

The parent unit or undo manager must accept the undo unit if *fCommit* is TRUE.

See Also

IOleUndoManager::Close

IOleParentUndoUnit::FindUnit Quick Info

Indicates if the specified unit is a child of this undo unit or one of its children, that is if the specified unit is part of the hierarchy in this parent unit.

```
HRESULT FindUnit(
  IOleUndoUnit* pUU          //Pointer to the undo unit to be found
);
```

Parameters

pUU

[in] Pointer to the undo unit to be found.

Return Values

S_OK

The specified undo unit is in the hierarchy subordinate to this parent.

S_FALSE

The specified undo unit is not part of the hierarchy under this parent. An error indicates an RPC failure condition.

Remarks

This is typically called by the undo manager in its implementation of its **DiscardFrom** method in the rare event that the unit being discarded is not a top-level unit. The parent unit should look in its own list first, then delegate to each child that is also a parent unit, as determined by doing a **QueryInterface** for **IOleParentUndoUnit**.

See Also

IOleUndoManager::DiscardFrom

IOleParentUndoUnit::GetParentState Quick Info

Returns state information about the innermost open parent undo unit.

```
HRESULT GetParentState(
  DWORD* pdwState          //Pointer to state information
);
```

Parameters

pdwState

[out] Pointer to state information. This information is a value taken from the **UASFLAGS** enumeration.

Return Values

S_OK

The parent's state was successfully returned.

Remarks

Note to Implementers

If the unit has an open child, it should delegate this method to that child. If not, it should fill in **pdwState* values appropriately and return. Note that a parent unit must never be blocked while it has an open child. If this happened it could prevent the child unit from being closed, which would cause serious problems.

Note to Callers

When checking for a normal state, use the UAS_MASK value to mask unused bits in the *pdwState* parameter to this method for future compatibility. For example:

```
fNormal = ((pdwState & UAS_MASK) == UAS_NORMAL)
```

See Also

UASFLAGS

IOleParentUndoUnit::Open Quick Info

Opens a new parent undo unit, which becomes part of the containing unit's undo stack.

```
HRESULT Open(
    IOleParentUndoUnit*     //Pointer to the parent undo unit to open
pPUU
);
```

Parameters

pPUU

[in] Pointer to the parent undo unit to be opened.

Return Values

S_OK

The parent undo unit was successfully opened or it is currently blocked.

Remarks

The specified parent unit is created and remains open. The undo manager then calls the **Add** or **Open** methods on this parent unit to add new units to it. This parent unit receives any additional undo units until its **Close** method is called.

The parent unit specified by *pPUU* is not added to the undo stack until its **Close** method is called with the *fCommit* parameter set to TRUE.

The parent undo unit or undo manager must contain any undo unit given to it unless it is blocked. If it is blocked, it must return S_OK but should do nothing else.

See Also

IOleUndoManager::Open

IOleUILinkContainer Quick Info

The **IOleUILinkContainer** interface is implemented by containers and used by OLE common dialog boxes. It supports these dialog boxes by providing the methods needed to manage a container's links.

The **IOleUILinkContainer** methods enumerate the links associated with a container, and specify how they should be updated, automatically or manually. They change the source of a link and obtain information associated with a link. They also open a link's source document, update links, and break a link to the source.

When to Implement

You must implement this interface if you are creating a container application that will use the Links, Change Source, or Update Links dialog boxes, as well as the Object Properties dialog box, which uses this interface indirectly. The Links dialog box calls back to the container application to perform OLE functions that manipulate the links within the container.

When to Use

OLE common dialog boxes use only this interface to manage the properties of a container's links. They can also use it to manage non-OLE (DDE and other container-specific) links.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleUILinkContainer Methods	Description
GetNextLink	Enumerates the links in the container.
SetLinkUpdateOptions	Sets update options.
GetLinkUpdateOptions	Determines current update options for the link.
SetLinkSource	Changes the source of the link.
GetLinkSource	Returns Links dialog box information about the link.
OpenLinkSource	Opens a link's source.
UpdateLink	Forces a link to connect to its source and update.
CancelLink	Disconnects selected links.

See Also

OleUIEditLinks, OleUIChangeSource, OleUIUpdateLinks, OleUIObjectProperties, OLEUIEDITLINKS

IOleUILinkContainer::CancelLink Quick Info

Disconnects the selected links.

```
HRESULT CancelLink(
  DWORD dwLink      //Unique 32-bit link identifier
);
```

Parameter

dwLink

[in] Container-defined unique 32-bit identifier for a single link. Containers can use the pointer to the link's container site for this value.

Return Values

This method supports the standard return values E_FAIL,

E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully disconnected the selected links.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Callers

Call **CancelLink** when the user selects the Break Link button from the Links dialog box. The link should be converted to a picture. The Links dialog box will not be dismissed for OLE links.

Notes To Implementers

For OLE links, **OleCreateStaticFromData** can be used to create a static picture object using the **IDataObject** interface of the link as the source.

See Also

IDataObject, **OleCreateStaticFromData**

IOleUILinkContainer::GetLinkSource Quick Info

Returns information about a link that can be displayed in the UI.

```
HRESULT GetLinkSource(
    DWORD dwLink,                   //Unique 32-bit link identifier
    LPTSTR FAR* lplpszDisplayName,  //Indirect pointer to length of display name portion
    ULONG FAR* lplenFileName,       //Pointer to length of file name portion
    LPTSTR FAR* lplpszFullLinkType, //Indirect pointer to full-link type string
    LPTSTR FAR* lplpszShortLinkType,//Indirect pointer to short-link type string
    BOOL FAR* lpfSourceAvailable,   //Pointer to availability of link
    BOOL FAR* lpfIsSelected         //Pointer to indicate that link entry should be selected in listbox
);
```

Parameters

dwLink

[in] Container-defined unique 32-bit identifier for a single link. See **IOleUILinkContainer::GetNextLink**.

lplpszDisplayName

[out] Indirect pointer to the allocated full-link source display name string. The Links dialog box will free this string.

lplenFileName

[out] Pointer to the length of the leading file name portion of the *lplpszDisplayName* string. If the link source is not stored in a file, then **lplenFileName* should be 0. For OLE links, call **IOleLink::GetSourceDisplayName**.

lplpszFullLinkType

[out] Indirect pointer to the allocated full-link type string that is displayed at the bottom of the Links dialog box. The Links dialog box will free this string. For OLE links, this should be the full User Type name. Use **IOleObject::GetUserType**, specifying USERCLASSTYPE_FULL for *dwFormOfType*.

lplpszShortLinkType

[out] Indirect pointer to the allocated short-link type string that is displayed in the listbox of the Links dialog box. The Links dialog box will free this string. For OLE links, this should be the short User Type name. Use **IOleObject::GetUserType**, specifying USERCLASSTYPE_SHORT for *dwFormOfType*.

lpfSourceAvailable

[out] Pointer that returns FALSE if it is known that a link is unavailable since the link is to some known but unavailable document. Certain options, such as Update Now, are disabled (grayed in the UI) for such cases.

lpfIsSelected

[out] Pointer to a BOOL variable that tells the Edit Links dialog box that this link's entry should be selected in the dialog's multi-selection listbox. **OleUIEditLinks** calls this method at least once for each item to be placed in the links list. If none of them return TRUE, then none of them will be selected when the dialog box is first displayed. If all of them return TRUE, then all will be displayed. That is, it returns TRUE if this link is currently part of the selection in the underlying document, FALSE if not. Any links that are selected in the underlying document are selected in the dialog box; this way,

the user can select a set of links and use the dialog box to update them or change their source(s) simultaneously.

Return Values

This method supports the standard return values E_FAIL,

E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully returned link information.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Callers

Call this method during dialog box initialization, after returning from the Change Source dialog box.

See Also

IOleLink::GetSourceDisplayName, **IOleObject::GetUserType**, **USERCLASSTYPE**, **OleUIChangeSource**, **OLEUICHANGESOURCE**_com_OLEUICHANGESOURCE_str

IOleUILinkContainer::GetLinkUpdateOptions Quick Info

Determine the current update options for a link.

```
HRESULT GetNextLink(
    DWORD dwLink,                   //Unique 32-bit link identifier
    DWORD FAR * lpdwUpdateOpt       //Pointer to address to return update option
);
```

Parameters

dwLink

[in] Container-defined unique 32-bit identifier for a single link. See **IOleUILinkContainer::GetNextLink**.

lpdwUpdateOpt

[out] Pointer to the location that the current update options will be written.

Return Values

This method supports the standard return values E_FAIL,

E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully determined update options.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Implementers

Containers can implement this method for OLE links simply by calling **IOleLink::SetUpdateOptions** on the link object.

See Also

IOleUILinkContainer::GetNextLink, **IOleUILinkContainer::SetLinkUpdateOptions**, **IOleLink::SetUpdateOptions**

IOleUILinkContainer::GetNextLink Quick Info

Enumerates the links in a container.

```
DWORD GetNextLink(
  DWORD dwLink      //Unique 32-bit linkidentifier
);
```

Parameter

dwLink

[in] Container-defined unique 32-bit identifier for a single link. This value is only passed to other methods on this interface, so it can be any value that uniquely identifies a link to the container. Containers frequently use the pointer to the link's container site object for this value.

Return Values

Returns a container's link identifiers in sequence; NULL if it has returned the last link.

Remarks

Notes to Callers

Call this method to enumerate the links in a container. If the value passed in *dwLink* is NULL, then the container should return the first link's 32-bit identifier. If *dwLink* identifies the last link in the container, then the container should return NULL.

See Also

IOleUILinkContainer::SetLinkUpdateOptions, **IOleUILinkContainer::GetLinkUpdateOptions**

IOleUILinkContainer::OpenLinkSource Quick Info

Opens the link's source.

```
HRESULT OpenLinkSource(
    DWORD dwLink      //Unique 32-bit link identifier
);
```

Parameter

dwLink

[in] Container-defined unique 32-bit identifier for a single link. Containers can use the pointer to the link's container site for this value.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully opened the link's source.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Callers

The **OpenLinkSource** method is called when the Open Source button is selected from the Links dialog box. For OLE links, call **IOleObject::DoVerb**, specifying OLEIVERB_SHOW for *iVerb*.

See Also

IOleObject::DoVerb, OLEVERB

IOleUILinkContainer::SetLinkSource Quick Info

Changes the source of a link.

```
HRESULT SetLinkSource(
  DWORD dwLink,                 //Unique 32-bit link identifier
  LPTSTR lpszDisplayName,       //Pointer to source string to parse
  ULONG FAR* lenFileName,       //Length of the file name portion
  ULONG FAR* pchEaten,          //Pointer to number of characters successfully parsed
  BOOL fValidateSource          //Specifies whether moniker should be validated
);
```

Parameters

dwLink

[in] Container-defined unique 32-bit identifier for a single link. See **IOleUILinkContainer::GetNextLink** .

lpszDisplayName

[in] Pointer to new source string to be parsed.

lenFileName

Length of the leading file name portion of the *lpszDisplayName* string. If the link source is not stored in a file, then *lenFileName* should be 0. For OLE links, call **IOleLink::GetSourceDisplayName**.

pchEaten

[out] Pointer to the number of characters successfully parsed in *lpszDisplayName*.

fValidateSource

[in] TRUE if the moniker should be validated; for OLE links, **MkParseDisplayName** should be called. FALSE if the moniker should not be validated. If possible, the link should accept the unvalidated source, and mark itself as unavailable.

Return Values

This method supports the standard return values E_FAIL,

E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully changed the links source.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Callers

Call this method from the Change Source dialog box, with *fValidateSource* initially set to TRUE. Change Source can be called directly or from the Links dialog box. If this call to **SetLinkSource** returns an error

(e.g., **MkParseDisplayName** failed because the source was unavailable), then you should display an Invalid Link Source message, and the user should be allowed to decide whether or not to fix the source. If the user chooses to fix the source, then the user should be returned to the Change Source dialog box with the invalid portion of the input string highlighted. If the user chooses not to fix the source, then **SetLinkSource** should be called a second time with *fValidateSource* set to FALSE, and the user should be returned to the Links dialog box with the link marked Unavailable.

See Also

MkParseDisplayName

IOleUILinkContainer::SetLinkUpdateOptions Quick Info

Sets a link's update options to Automatic (OLEUPDATE_ALWAYS) or Manual (OLEUPDATE_ONCALL).

```
HRESULT SetLinkUpdate(
  DWORD dwLink,          //Unique 32-bit link identifier
  DWORD dwUpdateOpt      //Update options
);
```

Parameters

dwLink

[in] Container-defined unique 32-bit identifier for a single link. See **IOleUILinkContainer::GetNextLink**.

dwUpdateOpt

[in] Update options, which can be Automatic (OLEUPDATE_ALWAYS) or Manual (OLEUPDATE_ONCALL).

Return Values

This method supports the standard return values E_FAIL,

E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully set the links update options.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

The user selects update options from the Links dialog box.

Notes To Implementers

Containers can implement this method for OLE links by simply calling **IOleLink::SetUpdateOptions** on the link object.

See Also

IOleUILinkContainer::GetNextLink, **IOleUILinkContainer::GetLinkUpdateOptions**, **IOleLink::SetUpdateOptions**

IOleUILinkContainer::UpdateLink Quick Info

Forces selected links to connect to their source and retrieve current information.

```
HRESULT UpdateLink(
  DWORD dwLink,           //Unique 32-bit link identifier
  DWORD fErrorMessage,    //Determines whether or not caller should display error message
  DWORD fReserved         //Reserved
);
```

Parameters

dwLink

[in] Container-defined unique 32-bit identifier for a single link. Containers can use the pointer to the link's container site for this value.

fErrorMessage

[in] Determines whether or not the caller (implementer of **IOleUILinkContainer**) should show an error message upon failure to update a link. The Update Links dialog box sets this to FALSE. The Object Properties and Links dialog boxes set it to TRUE.

fReserved

[in] Reserved for future use; must be set to FALSE.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully updated linked objects.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Callers

Call this method with *fErrorMessage* set to TRUE in cases where the user expressly presses a button to have a link updated, that is, presses the Links' Update Now button. Call it with FALSE in cases where the container should never display an error message, that is, where a large set of operations are being performed and the error should be propagated back to the user later, as might occur with the Update links progress meter. Rather than providing one message for each failure, assuming there are failures, provide a single message for all failures at the end of the operation.

Notes To Implementers

For OLE links, call **IOleObject::Update**.

See Also

IOleObject::Update

IOleUILinkInfo Quick Info

The **IOleUILinkInfo** interface is an extension of the **IOleUILinkContainer** interface. It returns the time that an object was last updated, which is link information that **IOleUILinkContainer** does not provide.

When To Implement

You must implement this interface so your container can support the "Link" page of the Object Properties dialog box. If you are writing a container that does not implement links, you do not need to implement this interface.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleUILinkContainer Methods	Description
GetNextLink	Enumerates the links in the container.
SetLinkUpdateOptions	Sets update options.
GetLinkUpdateOptions	Determines current update options for the link.
SetLinkSource	Changes the source of the link.
GetLinkSource	Returns Links dialog box information about link.
OpenLinkSource	Opens a link's source.
UpdateLink	Forces a link to connect to its source and update.
CancelLink	Breaks the link.

IOleUILinkInfo Methods	Description
GetLastUpdate	Determines the last time the object was updated, whether automatically or manually.

IOleUILinkInfo::GetLastUpdate Quick Info

Indicates when the object was last updated.

```
HRESULT GetLastUpdate(
    DWORD dwLink,                   //Unique 32-bit link identifier
    FILETIME FAR * lpLastUpdate     //Pointer to the time object was last updated
);
```

Parameters

dwLink

[in] Container-defined unique 32-bit identifier for a single link. Containers can use the pointer to the link's container site for this value.

lpLastUpdate

[out] Pointer to the time that the object was last updated.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully returned link information.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Implementers

If the time that the object was last updated is known, copy it to *lpLastUpdate*. If it is not known, then leave *lpLastUpdate* unchanged and Unknown will be displayed in the link page.

IOleUIObjInfo Quick Info

The **IOleUIObjInfo** interface is implemented by containers and used by the container's Object Properties dialog box and by the Convert dialog box. It provides information used by the General and View pages of the Object Properties dialog box , which display information about the object's size, location, type, and name. It also allows the object to be converted via the Convert dialog box. The View page allows the object's icon to be modified from its original form, and its display aspect to be changed (iconic versus content). Optionally, you can have your implementation of this interface allow the scale of the object to be changed.

When To Implement

You must implement this interface so your container application can support the **OleUIObjectProperties** function and the dialog box that it implements.

When To Use

Use this interface when you need to get and set information required by the Object Properties dialog box , and to support the Convert dialog box.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleUILinkContainer Methods	Description
GetNextLink	Enumerates the links in the container.
SetLinkUpdateOptions	Sets update options.
GetLinkUpdateOptions	Determines current update options for the link.
SetLinkSource	Changes the source of the link.
GetLinkSource	Returns Links dialog box information about link.
OpenLinkSource	Opens a link's source.
UpdateLink	Forces a link to connect to its source and update.
CancelLink	Breaks the link.

IOleUIObjInfo Methods	Description
GetObjectInfo	Gets general information about the object.
GetConvertInfo	Gets information that is used for the Convert dialog box.
ConvertObject	Converts the object once the user selects a destination type.
GetViewInfo	Gets the current icon, aspect, and scale of the object.

SetViewInfo	Sets the current icon, aspect, and scale of the object.

See Also

OleUIObjectProperties

IOleUIObjInfo::ConvertObject Quick Info

Converts the object to the type of the new CLSID.

```
HRESULT ConvertObject(
    DWORD dwObject,       //Unique 32-bit object identifier
    REFCLSID clsidNew     //CLSID to convert the object to
);
```

Parameters

dwObject

[in] Unique 32-bit identifier for the object.

clsidNew

[in] CLSID to convert the object to.

Return Values

This method supports the standard return values E_FAIL,

E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully returned link information.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Implementers

Your implementation of **ConvertObject** needs to convert the object to the CLSID specified. The actions taken by the convert operation are similar to the actions taken after calling **OleUIConvert**.

See Also

OleUIConvert

IOleUIObjInfo::GetConvertInfo Quick Info

Gets the conversion information associated with the specified object.

```
HRESULT GetConvertInfo(
  DWORD dwObject,                    //Unique 32-bit object identifier
  CLSID FAR * lpClassID,             //Pointer to location of CLSID of the object
  WORD FAR *lpwFormat,               //Pointer to clipboard format of the object
  CLSID FAR * lpConvertDefaultClassID,  //Pointer to default class to convert object to
  LPCLSID FAR * lplpClsidExclude,    //Indirect pointer to excluded CLSIDs
  UINT FAR * lpcClsidExclude         //Pointer to number of CLSIDs in lplpClsidExclude
);
```

Parameters

dwObject

[in] Unique 32-bit identifier for the object.

lpClassID

[out] Pointer to the location to return the object's CLSID.

lpwFormat

[out] Pointer to the clipboard format of the object.

lpConvertDefaultClassID

[out] Pointer to the default class, selected from the UI, to convert the object to.

lplpClsidExclude

[out] Indirect pointer to an array of CLSIDs that should be excluded from the UI for this object. May be NULL, if *lpcClsidExclude* is zero.

lpcClsidExclude

[out] Pointer to number of CLSIDs in *lplpClsidExclude*. May be zero.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully returned link information.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Implementers

You must fill in the CLSID of the object at a minimum. *lpwFormat* may be left at zero if the format of the storage is unknown.

IOleUIObjInfo::GetObjectInfo Quick Info

Gets size, type, name and location information about an object.

```
HRESULT GetObjectInfo(
  DWORD dwObject,                 //Unique 32-bit object identifier
  DWORD FAR *lpdwObjSize,         //Pointer to object's size
  LPTSTR FAR *lplpszLabel,        //Indirect pointer to object's label.
  LPTSTR FAR * lplpszType,        //Indirect pointer to object's "long" type
  LPTSTR FAR * lplpszShortType,   //Indirect pointer to object's "short" type
  LPTSTR FAR * lplpszLocation     //Indirect pointer to the object's source
);
```

Parameters

dwObject

[in] Unique 32-bit identifier for the object.

lpdwObjSize

[out] Pointer to the object's size, in bytes, on disk. This may be an approximate value.

lplpszLabel

[out] Indirect pointer to the object's label. May be NULL, which indicates that the implementation should not fill this in.

lplpszType

[out] Indirect pointer to the object's "long" type. May be NULL, which indicates that the implementation should not fill this in.

lplpszShortType

[out] Indirect pointer to the object's "short" type. May be NULL, which indicates that the implementation should not fill this in.

lplpszLocation

[out] Indirect pointer to the object's source. May be NULL, which indicates that the implementation should not fill this in.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully returned object information.

Remarks

The strings and the object's size are displayed in the object properties "General" page.

Notes To Implementers

Your implementation of **GetObjectInfo** should place each of the object's attributes in the out parameters provided. Set *lpdwObjSize* to (DWORD)-1 when the size of the object is unknown. Allocate all strings (the rest of the params) with the OLE task allocator obtained via **CoGetMalloc**, as is standard for all OLE interfaces with [out] string parameters, or you can simply use **CoTaskMemAlloc**.

See Also

CoGetMalloc, **CoTaskMemAlloc**

IOleUIObjInfo::GetViewInfo Quick Info

Gets the view information associated with the object.

```
HRESULT GetViewInfo(
  DWORD dwObject,                 //Unique 32-bit object identifier
  HGLOBAL FAR * phMetaPict,       //Pointer to object's current icon
  DWORD * pdvAspect,              ///Pointer to object's current aspect
  int * pnCurrentScale            ///Pointer to object's current scale
);
```

Parameters

dwObject

[in] Unique 32-bit identifier for the object.

phMetaPict

[in] Pointer to the object's current icon. Could be NULL, indicating that the caller is not interested in the object's current presentation.

pdvAspect

[in] Pointer to the object's current aspect. Could be NULL, indicating that the caller is not interested in the object's current aspect, i.e., DVASPECT_ICONIC or DVASPECT_CONTENT.

pnCurrentScale

[in] Pointer to the object's current scale. Could be NULL, indicating that the caller is not interested in the current scaling factor applied to the object in the container's view.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully returned link information.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Implementers

You must fill in the object's current icon, aspect, and scale.

See Also

OLEUIVIEWPROPS

IOleUIObjInfo::SetViewInfo Quick Info

Sets the view information associated with the object.

```
HRESULT SetViewInfo(
    DWORD dwObject,          //Unique 32-bit object identifier
    HGLOBAL hMetaPict,       //New icon for the object
    DWORD dvAspect,          //New display aspect for the objec
    int nCurrentScale,       //New scale for the objec
    BOOL bRelativeToOrig     //Scale of the object relative to origin
);
```

Parameters

dwObject

[in] Unique 32-bit identifier for the object.

hMetaPict

[in] New icon for the object.

dvAspect

[in] Object's new display aspect or view.

nCurrentScale

[in] Object's new scale.

bRelativeToOrig

[in] Scale of the object, relative to the origin. This value is TRUE if the new scale should be relative to the original scale of the object. If FALSE, *nCurrentScale* applies to the object's current size.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

Successfully returned link information.

E_ACCESSDENIED

Insufficient access permissions.

Remarks

Notes To Implementers

You should apply the new attributes (icon, aspect, and scale) to the object. If *bRelativeToOrig* is set to TRUE, *nCurrentScale* (in percentage units) applies to the original size of the object before it was scaled. If *bRelativeToOrig* is FALSE, *nCurrentScale* applies to the object's current size.

See Also

DVASPECT

IOleUndoManager Quick Info

The **IOleUndoManager** interface enables containers to implement multi-level undo and redo operations for actions that occur within contained controls.

The control must create an undo unit with the **IOleUndoUnit** interface or a parent undo unit with the **IOleParentUndoUnit** interface derived from **IOleUndoUnit**. Both of these interfaces perform the undo action and the parent undo unit additionally can contain nested undo units.

The undo manager provides a centralized undo and redo service. It manages parent undo units and simple undo units on the undo and redo stacks. Whether an object is UI-active or not, it can deposit undo units on these stacks by calling methods in the undo manager.

The centralized undo manager then has the data necessary to support the undo and redo user interface for the host application and can discard undo information gradually as the stack becomes full.

The undo manager is implemented as a service and objects obtain a pointer to **IOleUndoManger** from the **IServiceProvider** interface. It is usually implemented by the container. The service manages two stacks, the undo stack and the redo stack, each of which contains undo units generated by embedded objects or by the container application itself.

Undo units are typically generated in response to actions taken by the end user. An object does not generate undo actions for programmatic events. In fact, programmatic events should clear the undo stack since the programmatic event can possibly invalidate assumptions made by the undo units on the stack.

When the object's state changes, it creates an undo unit encapsulating all the information needed to undo that change. The object calls methods in the undo manager to place its undo units on the stack. Then, when the end user selects an Undo operation, the undo manager takes the top undo unit off the stack, invokes its action by calling its **IOleUndoUnit::Do** method, and then releases it. When an end user selects a Redo operation, the undo manager takes the top redo unit off the stack, invokes its action by calling its **IOleUndoUnit::Do** method, and then releases it.

The undo manager has three states: the base state, the undo state, and the redo state. It begins in the base state. To perform an action from the undo stack, it puts itself into the undo state, calls **IOleUndoUnit::Do** on the undo unit, and goes back to the base state. To perform an action from the redo stack, it puts itself into the redo state, calls **IOleUndoUnit::Do** on the undo unit, and goes back to the base state.

If the undo manager receives a new undo unit while in the base state, it places the unit on the undo stack and discards the entire redo stack. While it is in the undo state, it puts incoming units on the redo stack. While it is in the redo state, it places them on the undo stack without flushing the redo stack.

The undo manager also allows objects to discard the undo or redo stack starting from any object in either stack.

The host application determines the scope of an undo manager. For example, in one application, the scope might be at the document level; a separate undo manager is maintained for each document; and undo is managed independently for each document. However, another application maintain one undo manager, and therefore one undo scope, for the entire application.

Handling Errors

Having an undo operation fail and leaving the document in an unstable state is something the undo manager, undo units, and the application itself all have to work together to avoid. As a result, there are certain requirements that undo units, the undo manager, and the application or component using undo must conform to.

To perform an undo, the undo manager calls **IOleUndoUnit::Do** on one or more undo units which can, in turn, contain more units. If a unit somewhere in the hierarchy fails, the error will eventually reach the undo manager, which is responsible for making an attempt to roll back the state of the document to what it was before the call to the last top-level unit. The undo manager performs the rollback by calling **IOleUndoUnit::Do** on the unit that was added to the redo stack during the undo attempt. If the rollback also fails, then the undo manager is forced to abandon everything and return to the application. The undo manager indicates whether or not the rollback succeeded, and the application can take different actions based on this, such as reinitializing components so they're in a known state.

All the steps in adding an undo unit to the stack should be performed atomically. That is, all steps must succeed or none of them should succeed.

The host application that provides the undo manager decides what action to take when undo fails. At the very least, it should inform the user of the failure. The host application will be told by the undo manager whether or not the undo succeeded and whether or not the attempted rollback succeeded. In case both the undo and rollback failed, the host application can present the user with several options, including immediately shutting down the application.

Simple undo units must not change the state of any object if they return failure. This includes the state of the redo stack or undo stack if performing a redo. They are also required to put a corresponding unit on the redo or undo stack if they succeed. The application should be stable before and after the unit is called.

Parent undo units have the same requirements as simple units, with one exception. If one or more children succeeded prior to another child's failure, the parent unit must commit its corresponding unit on the redo stack and return the failure to its parent. If no children succeeded, the parent unit should commit its redo unit only if it has made a state change that needs to be rolled back. For example, suppose a parent unit contains three simple units. The first two succeed and added units to the redo stack, but the third one failed. At this point, the parent unit commits its redo unit and returns the failure.

As a side effect, the parent unit should never make state changes that depend on the success of their children. Doing this will cause the rollback behavior to break. If a parent unit makes state changes, it should do them before calling any children. Then, if the state change fails, it should not commit its redo unit, it should not call any children, and it should return the failure to its parent.

The undo manager has one primary requirement for error handling: to attempt rollback when an undo or redo fails.

Non-compliant Objects

Objects that do not support multi-level undo can cause serious problems for a global undo service. Since the object cannot be relied on to properly update the undo manager, any units submitted by other objects are also suspect, because their units may rely on the state of the non-compliant object. Attempting to undo a compliant object's units may not be successful, because the state in the non-compliant object will not match.

To detect objects that do not support multi-level undo, check for the OLEMISC_SUPPORTSMULTILEVELUNDO value. An object that can participate in the global undo service sets this value.

When an object without this value is added to a user-visible undo context, the safest thing to do is disable the undo user interface for this context. Alternatively, a dialog could be presented to the user, asking them whether to attempt to provide partial undo support, working around the non-compliance of the new object.

In addition, non-compliant objects may be added to nested containers. In this case, the nested container needs to notify the undo manager that undo can no longer be safely supported by calling **IOleUndoManager::Enable(FALSE)**.

When to Implement

Implement this interface to provide centralized undo services to the objects in a container.

When to Use

Call the methods in this interface to participate in global undo services.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleUndoManager Methods	Description
Open	Opens a new parent undo unit, which becomes part of its containing unit's undo stack.
Close	Closes the specified parent undo unit.
Add	Adds a simple undo unit to the collection.
GetOpenParentState	Returns state information about the innermost open parent undo unit.
DiscardFrom	Instructs the undo manager to discard the specified undo unit and all undo units below it on the undo or redo stack.
UndoTo	Instructs the undo manager to perform actions back through the undo stack, down to and including the specified undo unit.
RedoTo	Instructs the undo manager to invoke undo actions back through the redo stack, down to and including the specified undo unit.
EnumUndoable	Creates an enumerator object that the caller can use to iterate through a series of top-level undo units from the undo stack.
EnumRedoable	Creates an enumerator object that the caller can use to iterate through a series of top-level undo units from the redo stack.
GetLastUndoDescription	Returns the description for the top-level undo unit that is on top of the undo stack.
GetLastRedoDescription	Returns the description for the top-level undo unit that is on top of the redo stack.
Enable	Enables or disables the undo

manager.

See Also

IOleParentUndoUnit, IOleUndoUnit

IOleUndoManager::Add Quick Info

Adds a simple undo unit to the collection.

```
HRESULT Add(
  IOleUndoUnit* pUU          //Pointer to undo unit to be added
);
```

Parameters

pUU

[in] Pointer to undo unit to be added.

Return Values

S_OK

The specified unit was successfully added, the parent unit was blocked, or the undo manager is disabled.

Remarks

This method is implemented the same as **IOleParentUndoUnit::Add**. The parent undo unit or undo manager must accept any undo unit given to it, unless it is blocked. If it is blocked, it should do nothing but return S_OK.

Note to Implementers

If the undo manager is in the base state, it should put the new unit on the undo stack and discard the entire redo stack. If the undo manager is in the undo state, it should put new units on the redo stack. If the undo manager is in the redo state, it should put units on the undo stack without affecting the redo stack.

See Also

IOleParentUndoUnit::Add

IOleUndoManager::Close Quick Info

Closes the specified parent undo unit.

```
HRESULT Close(
    IOleParentUndoUnit*  //Pointer to the currently open parent unit
pPUU,
    BOOL fCommit         //Indicates whether to keep or discard the unit
);
```

Parameters

pPUU

[in] Pointer to the currently open parent unit to be closed.

fCommit

[in] Indicates whether to keep or discard the unit. If TRUE, the unit is kept in the collection. If FALSE, the unit is discarded. This parameter is used to allow the client to discard an undo unit under construction if an error or a cancellation occurs.

Return Values

S_OK

The parent unit had open child units and it was successfully closed. If the undo manager is disabled, it should immediately return S_OK and do nothing else.

S_FALSE

The parent undo unit did not have an open child and it was successfully closed.

E_INVALIDARG

If *pPUU* does not match the currently open parent undo unit, then implementations of this method should return E_INVALIDARG without changing any internal state unless the parent unit is blocked.

Remarks

This method is implemented the same as **IOleParentUndoUnit::Close**. A parent undo unit knows it is being closed when it returns S_FALSE from this method. At that time, it should terminate any communication with other objects which may be giving data to it through private interfaces.

Note to Implementers

To process a **Close**, a parent undo unit first checks to see if it has an open child unit. If it does not, it returns S_FALSE.

If it does have a child unit open, it calls the **Close** method on the child. If the child returns S_FALSE, then the parent undo unit verifies that *pPUU* points to the child unit, and closes that child undo unit. If the child returns S_OK then it handled the **Close** internally and its parent should do nothing.

If the parent unit is blocked, it should check the *pPUU* parameter to determine the appropriate return code. If *pPUU* is pointing to itself, then it should return S_FALSE.

Otherwise, it should return S_OK; the *fCommit* parameter is ignored; and no action is taken.

If *pPUU* does not match the currently open parent undo unit, then implementations of this method should return E_INVALIDARG without changing any internal state. The only exception to this is if the unit is blocked.

Note to Callers

An error return indicates a fatal error condition.

The parent unit or undo manager must accept the undo unit if *fCommit* is TRUE.

See Also

IOleParentUndoUnit::Close

IOleUndoManager::DiscardFrom Quick Info

Instructs the undo manager to discard the specified undo unit and all undo units below it on the undo or redo stack.

```
HRESULT DiscardFrom(
  IOleUndoUnit* pUU          //Pointer to undo unit to be discarded
);
```

Parameters

pUU

[in] Pointer to undo unit to be discarded. This parameter can be NULL to discard the entire undo or redo stack.

Return Values

S_OK

The specified undo unit was successfully discarded.

E_INVALIDARG

The specified undo unit was not found in the stacks.

E_UNEXPECTED

The undo manager is disabled.

Remarks

The undo manager first searches the undo stack for the given unit, and if not found there searches the redo stack. Once found, the given unit and all below it on the same stack are discarded. The undo unit may be a child of a parent unit contained by the undo manager, as determined by calling **IOleParentUndoUnit::FindUnit**. If it is a child unit, then the root unit containing the given unit and all units below it on the appropriate stack are discarded.

If there is an open parent unit and **DiscardFrom(NULL)** is called, the undo manager should immediately release and discard the open parent unit without calling the **Close** method first. When the object that opened the parent unit attempts to close it, **IOleUndoManager::Close** will return S_FALSE. If *pUU* is not NULL, then any open parent units should be left open.

See Also

IOleParentUndoUnit::FindUnit, **IOleUndoManager::Close**

IOleUndoManager::Enable Quick Info

Enables or disables the undo manager.

```
HRESULT Enable(
    BOOL fEnable        //Indicates whether to enable or disable the undo manager
);
```

Parameters

fEnable

[in] Indicates whether to enable or disable the undo manager. If TRUE, the undo manager should be enabled. If FALSE, the undo manager should be disabled.

Return Values

S_OK

The undo manager was successfully enabled or disabled.

E_UNEXPECTED

There is an open undo unit on the stack or the undo manager is currently performing an undo or a redo.

Remarks

The undo manager should clear both stacks when making the transition from enabled to disabled.

If the undo manager is disabled, each method in **IOleUndoManager** must behave as specified. See each method for details.

See Also

IOleUndoManager

IOleUndoManager::EnumRedoable Quick Info

Creates an enumerator object that the caller can use to iterate through a series of top-level undo units from the redo stack.

```
HRESULT EnumRedoable(
    IEnumOleUndoUnits**     //Indirect pointer to new enumerator object
ppEnum
);
```

Parameters

ppEnum

[out] Indirect pointer to the **IEnumOleUndoUnits** interface on the enumerator object.

Return Values

S_OK

The enumerator object was successfully created and the interface pointer was returned.

E_UNEXPECTED

The undo manager is disabled.

Remarks

A new enumerator object is created each time this method is called. If the series of enumerated items changes over time, the results of enumeration operations can vary from one call to the next.

This method calls **AddRef** on the new enumerator object to increment its reference count. The caller is responsible for calling **Release** on the enumerator object when it is no longer needed.

See Also

IEnumOleUndoUnits

IOleUndoManager::EnumUndoable Quick Info

Creates an enumerator object that the caller can use to iterate through a series of top-level undo units from the undo stack.

```
HRESULT EnumUndoable(
    IEnumOleUndoUnits**     //Indirect pointer to new enumerator object
ppEnum
);
```

Parameters

ppEnum

[out] Indirect pointer to the **IEnumOleUndoUnits** interface on the enumerator object.

Return Values

S_OK

The enumerator object was successfully created and the interface pointer was returned.

E_UNEXPECTED

The undo manager is disabled.

Remarks

A new enumerator object is created each time this method is called. If the series of enumerated items changes over time, the results of enumeration operations can vary from one call to the next.

This method calls **AddRef** on the new enumerator object to increment its reference count. The caller is responsible for calling **Release** on the enumerator object when it is no longer needed.

See Also

IEnumOleUndoUnits

IOleUndoManager::GetLastRedoDescription Quick Info

Returns the description for the top-level undo unit that is on top of the redo stack.

```
HRESULT GetLastRedoDescription(
  BSTR* pBstr                    //Pointer to string
);
```

Parameters

pBstr

[out] Pointer to a string that contains a description of the top-level undo unit on the redo stack.

Return Values

S_OK

The string was successfully returned and it contains a valid description.

E_FAIL

The undo stack is empty.

E_UNEXPECTED

The undo manager is disabled.

Remarks

This method provides a convenient shortcut for the host application to add a description to its Edit Redo menu item. The **pbstr* parameter is a string allocated with the standard string allocator. The caller is responsible for freeing this string.

See Also

IOleUndoManager::GetLastUndoDescription

IOleUndoManager::GetLastUndoDescription Quick Info

Returns the description for the top-level undo unit that is on top of the undo stack.

```
HRESULT GetLastUndoDescription(
    BSTR* pBstr                    //Pointer to string
);
```

Parameters

pBstr

[out] Pointer to a string that contains a description of the top-level undo unit on the undo stack.

Return Values

S_OK

The string was successfully returned and it contains a valid description.

E_FAIL

The undo stack is empty.

E_UNEXPECTED

The undo manager is disabled.

Remarks

This method provides a convenient shortcut for the host application to add a description to its Edit Undo menu item. The **pbstr* parameter is a string allocated with the standard string allocator. The caller is responsible for freeing this string.

See Also

IOleUndoManager::GetLastRedoDescription

IOleUndoManager::GetOpenParentState Quick Info

Returns state information about the innermost open parent undo unit.

```
HRESULT GetOpenParentState(
  DWORD* pdwState          //Pointer to state information
);
```

Parameters

pdwState

[out] Pointer to state information. This information is a value taken from the **UASFLAGS** enumeration.

Return Values

S_OK

There was an open parent unit and its state was successfully returned or the undo manager is disabled.

S_FALSE

There is no open parent unit.

Remarks

Note to Implementers

If there is an open parent unit, this method calls **IOleParentUnit::GetParentState**.

If the undo manager is disabled, it should fill the *pdwState* parameter with UAS_BLOCKED and return S_OK.

Note to Callers

When checking for a normal state, use the UAS_MASK value to mask unused bits in the *pdwState* parameter to this method for future compatibility. For example:

```
fNormal = ((pdwState & UAS_MASK) == UAS_NORMAL)
```

See Also

UASFLAGS

IOleUndoManager::Open Quick Info

Opens a new parent undo unit, which becomes part of its containing unit's undo stack.

```
HRESULT Open(
  IOleParentUndoUnit* pPUU    //Pointer to the parent undo unit to open
);
```

Parameters

pPUU

[in] Pointer to the parent undo unit to be opened.

Return Values

S_OK

The parent undo unit was successfully opened or if a currently open unit is blocked. If the undo manager is currently disabled, it should return S_OK and do nothing else.

Remarks

This method is implemented the same as **IOleParentUndoUnit::Open**. The specified parent unit is created and remains open. The undo manager then calls the **Add** or **Open** methods on this parent unit to add new units to it. This parent unit receives any additional undo units until its **Close** method is called.

The parent unit specified by *pPUU* is not added to the undo stack until its **Close** method is called with the *fCommit* parameter set to TRUE.

The parent undo unit or undo manager must contain any undo unit given to it unless it is blocked. If it is blocked, it must return S_OK but should do nothing else.

See Also

IOleParentUndoUnit::Open

IOleUndoManager::RedoTo

Instructs the undo manager to invoke undo actions back through the redo stack, down to and including the specified undo unit.

```
HRESULT RedoTo(
    IOleUndoUnit* pUU          //Pointer to the top level unit to redo
);
```

Parameters

pUU

[in] [in] Pointer to the top level unit to redo. If this parameter is NULL, the most recently added top level unit is used.

Return Values

S_OK

The specified undo unit was successfully invoked to perform its undo action.

E_INVALIDARG

The specified undo unit is not on the redo stack.

E_ABORT

Both the undo attempt and the rollback attempt failed. The undo manager should never propagate the E_ABORT obtained from a contained undo unit. Instead, it should map any E_ABORT values returned from other undo units to E_FAIL.

E_UNEXPECTED

The undo manager is disabled.

Remarks

This method calls the **IOleUndoUnit::Do** method on each top-level undo unit. Then, it releases that undo unit.

Note that the specified undo unit must be a top-level unit, typically retrieved through **IOleUndoManager::EnumRedoable**.

In case an error is returned from the undo unit, the undo manager needs to attempt to rollback the state of the document to recover from the error by performing actions on the redo stack.

No matter what the success of the rollback, the undo manager should always clear both stacks before returning the error.

If the undo manager has called the **Do** method on more than one top-level unit, it should only rollback the unit that returned the error. The top-level units that succeeded should not be rolled back.

The undo manager must also keep track of whether or not units were added to the opposite stack so it won't attempt rollback if nothing was added. See the **IOleUndoManager** interface for detailed description of error handling.

See Also

IOleUndoManager::EnumRedoable, IOleUndoManager::UndoTo, IOleUndoUnit::Do

IOleUndoManager::UndoTo Quick Info

Instructs the undo manager to invoke undo actions back through the undo stack, down to and including the specified undo unit.

```
HRESULT UndoTo(
  IOleUndoUnit* pUU          //Pointer to the top level unit to undo
);
```

Parameters

pUU

[in] Pointer to the top level unit to undo. If this parameter is NULL, the most recently added top level unit is used.

Return Values

S_OK

The specified undo unit was successfully invoked to perform its undo action.

E_INVALIDARG

The specified undo unit is not on the undo stack.

E_ABORT

Both the undo attempt and the rollback attempt failed. The undo manager should never propagate the E_ABORT obtained from a contained undo unit. Instead, it should map any E_ABORT values returned from other undo units to E_FAIL.

E_UNEXPECTED

The undo manager is disabled.

Remarks

This method calls the **IOleUndoUnit::Do** method on each top-level undo unit. Then, it releases that undo unit.

Note that the specified undo unit must be a top-level unit, typically retrieved through **IOleUndoManager::EnumUndoable**.

In case an error is returned from the undo unit, the undo manager needs to attempt to rollback the state of the document to recover from the error by performing actions on the redo stack.

No matter what the success of the rollback, the undo manager should always clear both stacks before returning the error.

If the undo manager has called the **Do** method on more than one top-level unit, it should only rollback the unit that returned the error. The top-level units that succeeded should not be rolled back.

The undo manager must also keep track of whether or not units were added to the opposite stack so it won't attempt rollback if nothing was added. See the **IOleUndoManager** interface for detailed description of error handling.

See Also

IOleUndoManager::EnumUndoable, IOleUndoManager::RedoTo, IOleUndoUnit::Do

IOleUndoUnit Quick Info

The **IOleUndoUnit** interface is the main interface on an undo unit. An undo unit encapsulates the information necessary to undo or redo a single action.

When an object's state changes and it needs to create an undo unit, it first needs to know what parent units are open. It calls the **IOleUndoManager::GetOpenParentState** method to determine this. If the call returns S_FALSE, then there is no enabling parent. If the call returns S_OK but the UAS_NOPARENTENABLE flag is set, then the open parent is a disabling parent. In either of these cases, the object calls **IOleUndoManager::DiscardFrom(NULL)** on the undo manager and skips creating the undo unit.

If the method returns S_OK, but the UAS_BLOCKED flag is set, then the open parent is a blocking parent. The object does not need to create an undo unit, since it would be immediately discarded. If the return value is S_OK and neither of the bit flags are set, then the object creates the undo unit and calls **IOleUndoManager::Add** on the undo manager.

The object should retain a pointer to the undo manager.

When to Implement

An object creates an undo unit that implements this interface when the end user has performed an action that can be undone. The object calls the **IOleUndoManager::Add** method to add the undo unit to the undo stack. Most controls can implement this interface to support the centralized undo operations. They do not have to implement **IOleParentUndoUnit** or **IOleUndoManager**. The undo manager with the **IOleUndoManager** interface is usually provided by the object container.

When to Use

The undo manager calls the **Do** and **GetDescription** methods in this interface to perform undo actions and to get strings that can be displayed in the user interface to describe the undo action. A parent undo unit can call the **GetUnitType** and **OnNextAdd** methods.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleUndoUnit Methods	Description
Do	Instructs the undo unit to carry out its action.
GetDescription	Returns a string that describes the undo unit and can be used in the undo or redo user interface.
GetUnitType	Returns the CLSID and a type identifier for the undo unit.
OnNextAdd	Notifies the last undo unit in the collection that a new unit has been added.

See Also

IOleParentUndoUnit, IOleUndoManager

IOleUndoUnit::Do Quick Info

Instructs the undo unit to carry out its action. Note that if it contains child undo units, it must call their **Do** methods as well.

```
HRESULT Do(
  IOleUndoManager* pUndoManager    //Pointer to the undo manager
);
```

Parameters

pUndoManager

[in] Pointer to the undo manager.

Return Values

S_OK

The undo unit successfully carried out its action.

Remarks

The undo unit is responsible for carrying out its action. Performing its own undo action results in another action that can potentially be reversed. However, if *pUndoManager* is NULL, the undo unit should perform its undo action but should not attempt to put anything on the redo or undo stack.

If *pUndoManager* is not NULL, then the unit is required to put a corresponding unit on the redo or undo stack. As a result, this method either moves itself to the redo or undo stack, or it creates a new undo unit and adds it to the appropriate stack. After creating a new undo unit, this undo unit calls **IOleUndoManager::Open** or **IOleUndoManager::Add**. The undo manager will put the new undo unit on the undo or redo stack depending on its current state.

A parent unit must pass to its children the same undo manager, possibly NULL, that was given to the parent. It is permissible, but not necessary, when *pUndoManager* is NULL to open a parent unit on the redo or undo stack as long as it is not committed. A blocked parent unit ensures that nothing is added to the stack by child units.

If this undo unit is a parent unit, it should put itself on the redo or undo stack before calling the **Do** on its children.

After calling this method, the undo manager must release the undo unit.

Note to Implementers

See the **IOleUndoManager** interface for error handling strategies for undo units. The error handling strategy affects the implementation of this method, particularly for parent units.

See Also

IOleUndoManager::Add, **IOleUndoManager::Open**

IOleUndoUnit::GetDescription Quick Info

Returns a string that describes the undo unit and can be used in the undo or redo user interface.

```
HRESULT GetDescription(
    BSTR* pbstr                    //Pointer to string
);
```

Parameters

pbstr

[out] Pointer to string describing this undo unit.

Return Values

S_OK

The string was successfully returned.

Remarks

All units are required to provide a user-readable description of themselves.

Note to Callers

The **pbstr* parameter is allocated with the standard string allocator. The caller is responsible for freeing this string.

IOleUndoUnit::GetUnitType Quick Info

Returns the CLSID and a type identifier for the undo unit.

```
HRESULT GetUnitType(
  CLSID* pClsid ,          //Pointer to CLSID for undo unit
  LONG* plID               //Pointer to type identifier for undo unit
);
```

Parameters

pclsid

[out] Pointer to CLSID for the undo unit.

plID

[out] Pointer to the type identifier for the undo unit.

Return Values

S_OK

Both the CLSID and type identifier were successfully returned.

Remarks

A parent undo unit can call this method on its child units to determine whether it can apply special handling to them. The CLSID returned can be the CLSID of the undo unit itself, of its creating object, or an arbitrary GUID. The undo unit has the option of returning CLSID_NULL, in which case the caller can make no assumptions about the type of this unit. The only requirement is that the CLSID and type identifier together uniquely identify this type of undo unit.

Note that the undo manager and parent undo units do not have the option of accepting or rejecting child units based on their type.

IOleUndoUnit::OnNextAdd Quick Info

Notifies the last undo unit in the collection that a new unit has been added.

HRESULT OnNextAdd(void);

Return Values

S_OK

Implementations of this method always return S_OK. The HRESULT return type is provided only for remotability.

Remarks

An object can create an undo unit for an action and add it to the undo manager but can continue inserting data into it through private interfaces. When the undo unit receives a call to this method, it communicates back to the creating object that the context has changed. Then, the creating object stops inserting data into the undo unit.

The parent undo unit calls this method on its most recently added child undo unit to notify the child unit that the context has changed and a new undo unit has been added.

For example, this method is used for supporting fuzzy actions, like typing, which do not have a clear point of termination but instead are terminated only when something else happens.

This method may not always be called if the undo manager or an open parent unit chooses to discard the unit by calling **Release** instead. Any connection which feeds data to the undo unit behind the scenes through private interfaces should not **AddRef** the undo unit.

Note to Implementers

Note that parent units merely delegate this method to their most recently added child unit. A parent unit should terminate communication through any private interfaces when it is closed. A parent unit knows it is being closed when it receives S_FALSE from calling **IOleParentUndoUnit::Close**.

See Also

IOleParentUndoUnit::Close

IOleWindow Quick Info

The **IOleWindow** interface provides methods that allow an application to obtain the handle to the various windows that participate in in-place activation, and also to enter and exit context-sensitive help mode.

Several other in-place activation interfaces are derived from the **IOleWindow** interface. Containers and objects must implement and use these interfaces in order to support in-place activation. The following table briefly summarizes each of these interfaces:

IOleWindow	The base interface. Implemented and used by containers and objects to obtain window handles and manage context-sensitive help.
IOleInPlaceObject	Implemented by objects and used by an object's immediate container to activate and deactivate the object.
IOleInPlaceActiveObject	Implemented by objects and used by the top-level container to manipulate the object while it is active. Provides a direct channel of communication between an active object and its frame and document windows.
IOleInPlaceUIWindow	Implemented by containers and used by objects to manipulate the container's document window.
IOleInPlaceFrame	Implemented by containers and used by objects to control the container's frame window.
IOleInPlaceSite	Implemented by containers and used by objects to interact with the in-place client site.
IOleInPlaceSiteEx	Implemented by containers and called by objects to optimize activation and deactivation.
IOleInPlaceSiteWIndowless	Implemented by containers and called by windowless object to obtain services from its container.
IOleInPlaceObjectWindowless	Implemented by windowless objects called by containers to support window message processing and drag and drop operations for windowless objects.

These interfaces can be arranged in three hierarchical levels with various interfaces implemented at each level. Calls that install user-interface menus commands and frame adornments, activate and switch between windows, and dispatch menu and keystrokes take place between the top-level container and the active object. Calls that support activating, deactivating, scrolling, or clipping span the containment hierarchy, with each level performing the correct actions.

{ewc msdncd, EWGraphic, bsd23521 0 /a "SDK.WMF"}

When to Implement

The inherited methods of this interface are implemented by all in-place objects and containers.

When to Use

Use this interface to obtain the window handle to the windows associated with in-place activation (frame, document, parent, and in-place object). It is also used to enter and exit context-sensitive help.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IOleWindow Methods	Description
GetWindow	Gets a window handle.
ContextSensitiveHelp	Controls enabling of context-sensitive help.

See Also

IOleInPlaceObject, IOleInPlaceActiveObject, IOleInPlaceUIWindow, IOleInPlaceFrame, IOleInPlaceSite, OleCreateMenuDescriptor, OleDestroyMenuDescriptor, OleTranslateAccelerator, IOleInPlaceSiteEx, IOleInPlaceSiteWindowless, IOleInPlaceObjectWindowless

IOleWindow::ContextSensitiveHelp Quick Info

Determines whether context-sensitive help mode should be entered during an in-place activation session.

```
HRESULT ContextSensitiveHelp(
  BOOL fEnterMode      //Specifies whether or not to enter help mode
);
```

Parameter

fEnterMode

[in] TRUE if help mode should be entered; FALSE if it should be exited.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The help mode was entered or exited successfully, depending on the value passed in *fEnterMode*.

Remarks

Applications can invoke context-sensitive help when the user

- Presses SHIFT+F1, then clicks a topic.
- Presses F1 when a menu item is selected.

When SHIFT+F1 is pressed, either the frame or active object can receive the keystrokes. If the container's frame receives the keystrokes, it calls its containing document's **IOleWindow::ContextSensitiveHelp** method with *fEnterMode* set to TRUE. This propagates the help state to all of its in-place objects so they can correctly handle the mouse click or WM_COMMAND.

If an active object receives the SHIFT+F1 keystrokes, it calls the container's **IOleInPlaceSite::ContextSensitiveHelp** method with *fEnterMode* TRUE, which then recursively calls each of its in-place sites until there are no more to be notified. The container then calls its document's or frame's **ContextSensitiveHelp** method with *fEnterMode* TRUE.

When in context-sensitive help mode, an object that receives the mouse click can either:

1. Ignore the click if it does not support context-sensitive help, or
2. Tell all the other objects to exit context-sensitive help mode with **ContextSensitiveHelp** set to FALSE and then provide help for that context.

An object in context-sensitive help mode that receives a WM_COMMAND should tell all the other in-place objects to exit context-sensitive help mode and then provide help for the command.

If a container application is to support context-sensitive help on menu items, it must either provide its own message filter so that it can intercept the F1 key or ask the OLE library to add a message filter by calling **OleSetMenuDescriptor**, passing valid, non-NULL values for the *lpFrame* and *lpActiveObj* parameters.

See Also

OleSetMenuDescriptor

IOleWindow::GetWindow Quick Info

Returns the window handle to one of the windows participating in in-place activation (frame, document, parent, or in-place object window).

```
HRESULT GetWindow(
  HWND * phwnd      //Pointer to where to return window handle
);
```

Parameter

phwnd

[out] Pointer to where to return the window handle.

Return Values

This method supports the standard return values E_FAIL, E_OUTOFMEMORY, E_INVALIDARG, and E_UNEXPECTED, as well as the following:

S_OK

The window handle was successfully returned.

Note For windowless objects, this method should always fail and return E_FAIL.

Remarks

Five types of windows comprise the windows hierarchy. When a object is active in place, it has access to some or all of these windows:

Window	Description
Frame	The outermost main window where the container application's main menu resides.
Document	The window that displays the compound document containing the embedded object to the user.
Pane	The subwindow of the document window that contains the object's view. Applicable only for applications with split-pane windows.
Parent	The container window that contains that object's view. The object application installs its window as a child of this window.
In-place	The window containing the active in-place object. The object application creates this window and installs it as a child of its hatch window, which is a child of the container's parent window.

Each type of window has a different role in the in-place activation architecture. However, it is not necessary to employ a separate physical window for each type. Many container applications use the same window for their frame, document, pane, and parent windows.

IParseDisplayName Quick Info

The **IParseDisplayName** interface parses a displayable name string to convert it into a moniker. Display name parsing is necessary when the end user inputs a string to identify a component, as in the following situations:

- A compound document application that supports linked components typically supports the Edit:Links... dialog box. Through this dialog box, the end user can enter a display name to specify a new link source for a specified linked component. The compound document needs to have this input string converted into a moniker.
- A script language such as the macro language of a spreadsheet can allow textual references to a component. The language's interpreter needs to have such a reference converted into a moniker in order to execute the macro.

When to Implement

Compound document applications that support links to embedded components or to pseudo-objects within their documents must provide an implementation of the **IOleItemContainer** interface, which is derived indirectly from **IParseDisplayName**. In effect, such a compound document is providing a namespace for identifying its internal components; and its **IOleItemContainer** implementation (which includes the **IParseDisplayName** implementation) is the interface through which another application can access this namespace. Alternatively, the compound document application can implement **IParseDisplayName** as part of its class object, which is accessible through the **CoGetClassObject** function.

When to Use

If you are implementing your own moniker class, you might need to use this interface from your implementation of **IMoniker::ParseDisplayName**. If you call the **MkParseDisplayName** function, you are indirectly using **IParseDisplayName**.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IParseDisplayName Method	Description
ParseDisplayName	Parses the display name.

See Also

IMoniker::ParseDisplayName, IOleItemContainer, MkParseDisplayName

IParseDisplayName::ParseDisplayName Quick Info

Parses the display name to extract a component of the string that it can convert into a moniker, using the maximum number of characters from the left side of the string.

```
HRESULT ParseDisplayName(
  IBindCtx *pbc,                //Pointer to bind context
  LPOLESTR pszDisplayName,      //Pointer to string containing display name
  ULONG *pchEaten,              //Pointer to length, in characters, of display name
  IMoniker **ppmkOut            //Indirect pointer to moniker that results
);
```

Parameters

pbc

[in] Pointer to the bind context to be used in this binding operation.

pszDisplayName

[in] Pointer to a zero-terminated string containing the display name to be parsed. For Win32 applications, the **LPOLESTR** type indicates a wide character string (two bytes per character); otherwise, the string has one byte per character.

pchEaten

[out Pointer to the number of characters in the display name that correspond to the *ppmkOut* moniker.

ppmkOut

[out] Indirect pointer to the resulting moniker. If an error occurs, the implementation sets **ppmkOut* to NULL. If **ppmkOut* is non-NULL, the implementation must call *(*ppmkOut)->***IUnknown::AddRef**; so it is the caller's responsibility to call *(*ppmkOut)->***IUnknown::Release**.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

Success.

MK_E_SYNTAX

Syntax error in the display name.

MK_E_NOOBJECT

The display name does not identify a component in this namespace.

See Also

MkParseDisplayName, **IMoniker::ParseDisplayName**

IPerPropertyBrowsing Quick Info

The **IPerPropertyBrowsing** interface accesses the information in the property pages offered by an object.

When to Implement

Implement this interface on all objects that have property pages so that clients can access information about the properties.

When to Use

Use this interface to access information about an object's properties.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPerPropertyBrowsing Methods	Description
GetDisplayString	Returns a text string describing the specified property.
MapPropertyToPage	Returns the CLSID of the property page that allows manipulation of the specified property.
GetPredefinedStrings	Returns a counted array of strings (LPOLESTR pointers) listing the descriptions of the allowable values that the specified property can accept (i.e., the values returned from **IPerPropertyBrowsing::GetPredefinedValue**.
GetPredefinedValue	Returns a VARIANT containing the value of a property identified with *dispID* that is associated with a predefined string name as returned from **IPerPropertyBrowsing::GetPredefinedStrings**.

See Also

IPropertyPage, IPropertyPage2, IPropertyPageSite, ISpecifyPropertyPages

IPerPropertyBrowsing::GetDisplayString Quick Info

Returns a text string describing the property identified with *dispID* in the caller's user interface. In other words, the returned text is a displayable name describing the property and can be displayed in the caller's user interface. The string itself is a BSTR allocated by this method with **SysAllocString**. Upon return, the string is the responsibility of the caller, which must free it with **SysFreeString** when it is no longer needed.

```
HRESULT GetDisplayString(

  DISPID dispID ,      //Dispatch identifier for the property
  BSTR *pbstr          //Receives a pointer to the displayable string describing the property
);
```

Parameters

dispID

[in] Dispatch identifier of the property whose display name is requested.

pbstr

[out] Pointer to the BSTR containing the display name for the property identified with *dispID*.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The display name was successfully returned.

E_NOTIMPL

The object does not provide display names for individual properties. The caller has the recourse to check the object's type information for the text name of the object in this case.

E_POINTER

The address in *pbstr* is not valid. For example, it may be NULL.

Remarks

Notes to Implementers

The caller is responsible for freeing the *pbstr* string with **SysFreeString** when it is no longer needed.

IPerPropertyBrowsing::GetPredefinedStrings Quick Info

Returns a counted array of string pointers (LPOLESTR pointers). The strings pointed to provide a list of names that each correspond to values that the property specified with *dispID* can accept.

```
HRESULT GetPredefinedStrings(
  DISPID dispID ,                 //Dispatch identifier for property
  CALPOLESTR *pcaStringsOut ,     //Receives a pointer to an array of strings
  CADWORD *pcaCookiesOut          //Receives a pointer to array of DWORDs
);
```

Parameters

dispID

[in] Dispatch identifier of the property for which the caller is requesting the string list.

pcaStringsOut

[out] Pointer to a caller-allocated, counted array structure that contains the element count and address of a method-allocated array of string pointers. This method also allocates memory for the string values containing the predefined names, and it stores the string pointers in the array. If the method fails, no memory is allocated, and the contents of the structure are undefined.

pcaCookiesOut

[out] Pointer to the caller-allocated, counted array structure that contains the element count and address of a method-allocated array of DWORDs. The DWORD values in the array can be passed to **IPerPropertyBrowsing::GetPredefinedValue** to retrieve the value associated with the name in the same array position inside *pcaStringsOut*. If the method fails, no memory is allocated, and the contents of the structure are undefined.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The arrays were allocated and filled successfully.

E_NOTIMPL

This method is not implemented and predefined names are not supported.

E_POINTER

The address in *pcaStringsOut* or *pcaCookiesOut* is not valid. For example, either may be NULL.

Remarks

Each string returned in the array pointed to by *pcaStringsOut* has a matching token in the counted array pointed to by *pcaCookiesOut*, where the token can be passed to **IPerPropertyBrowsing::GetPredefinedValue** to get the actual value (a VARIANT) corresponding to the string.

Using the predefined strings, a caller can obtain a list of strings for populating user interface elements,

such as a drop-down listbox. When the end user selects one of these strings as a value to assign to a property, the caller can then obtain the corresponding value through **IPerPropertyBrowsing::GetPredefinedValue**.

Notes to Callers

Both the **CALPOLESTR** and **CADWORD** structures passed to this method are caller-allocated. The caller is responsible for freeing each string pointed to from the **CALPOLESTR** array as well as the **CALPOLESTR** structure.

All memory is allocated with **CoTaskMemAlloc**. The caller is responsible for freeing the strings and the array with **CoTaskMemFree**.

Upon return from this method, the caller is responsible for all this memory and must free it when it is no longer needed. Code to achieve this appears as follows:

```
CALPOLESTR      castr;
CWDWORD         cadw;
ULONG           i;

pIPerPropertyBrowsing->GetPredefinedStrings(dispID, &castr, &cadw);

//...Use the strings and the cookies

CoTaskMemFree((void *)cadw.pElems);

for (i=0; i < castr.cElems; i++)
    CoTaskMemFree((void *)castr.pElems[i]);

CoTaskMemFree((void *)castr.pElems);
```

Notes to Implementers

Support for predefined names and values is not required. If your object does not support these names, return E_NOTIMPL from this method. If this method is not implemented, **IPerPropertyBrowsing::GetPredefinedValue** must not be implemented either.

This method fills the *cElems* and *pElems* fields of the **CADWORD** and **CALPOLESTR** structures. It allocates the arrays pointed to by these structures with **CoTaskMemAlloc** and fills those arrays. In the **CALPOLESTR** case, this method also allocates each string with **CoTaskMemAlloc**, storing each string pointer in the array.

See Also

CADWORD, **CALPOLESTR**, **CoTaskMemAlloc**, **CoTaskMemFree**, **IPerPropertyBrowsing::GetPredefinedValue**

IPerPropertyBrowsing::GetPredefinedValue Quick Info

Returns a VARIANT containing the value of the property specified by *dispID*. This property is associated with a predefined string name as returned from **IPerPropertyBrowsing::GetPredefinedStrings**. The predefined string is identified by a token returned from **GetPredefinedStrings**.

```
HRESULT GetPredefinedValue(
  DISPID dispID ,          //Dispatch identifier for property
  DWORD dwCookie ,         //Token returned
  VARIANT *pVarOut         //Receives a pointer to a VARIANT value for the property
);
```

Parameters

dispID

[in] Dispatch identifier of the property for which a predefined value is requested.

dwCookie

[in] Token identifying which value to return. The token was previously returned in the *pcaCookiesOut* array filled by **IPerPropertyBrowsing::GetPredefinedStrings**.

pVarOut

[out] Pointer to the VARIANT value for the property.

Return Values

This method supports the standard return values E_INVALIDARG,

E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The value was returned successfully.

E_NOTIMPL

This object does not support predefined strings or predefined values.

E_POINTER

The address in *pVarOut* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

The caller is responsible for freeing any allocations contained in the VARIANT. Unless the *vt* field of **VARIANT** is VT_VARIANT, the caller can free memory using a single call to **VariantClear**. Otherwise, the caller must recursively free the contained VARIANTs before freeing the outer VARIANT.

Notes to Implementers

Support for predefined names and values is not required. If your object does not support these names, return E_NOTIMPL from this method. If this method is not implemented, **IPerPropertyBrowsing::GetPredefinedStrings** must not be implemented either.

This method allocates any memory needed inside the VARIANT.

See Also

IPerPropertyBrowsing::GetPredefinedStrings

IPerPropertyBrowsing::MapPropertyToPage Quick Info

Returns the CLSID of the property page associated with the specified property. In other words, this method maps a specified property to the property page that allows a user to manipulate that property. The CLSID returned from this method can be passed to **OleCreatePropertyFrameIndirect** to specify the initial page to display in the property sheet.

```
HRESULT MapPropertyToPage(
  DISPID dispID ,      //Dispatch identifier for the property
  CLSID *pclsid        //Receives a pointer to CLSID for property
);
```

Parameters

dispID

[in] Dispatch identifier of the property of interest.

pclsid

[out] Pointer to the CLSID identifying the property page associated with the property specified by *dispID*. If this method fails, **pclsid* is set to CLSID_NULL.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The CLSID was successfully returned.

E_NOTIMPL

The object does not support property pages at all or doesn't support mapping properties to the page CLSID. In other words, this feature of specific property browsing is not supported.

E_POINTER

The address in *pclsid* is not valid. For example, it may be NULL.

IPersist

Quick Info

The **IPersist** interface defines the single method **GetClassID**, which is designed to supply the CLSID of an object that can be stored persistently in the system. A call to this method can allow the object to specify which object handler to use in the client process, as it is used in the OLE default implementation of marshaling.

IPersist is the base interface for three other interfaces: **IPersistStorage**, **IPersistStream**, and **IPersistFile**. Each of these interfaces, therefore, includes the **GetClassID** method, and the appropriate one of these three interfaces is implemented on objects that can be serialized to a storage, a stream, or a file. The methods of these interfaces allow the state of these objects to be saved for later instantiations, and load the object using the saved state. Typically, the persistence interfaces are implemented by an embedded or linked object, and are called by the container application or the default object handler

When to Implement

You must implement the single method of **IPersist** in implementing any one of the other persistence interfaces: **IPersistStorage**, **IPersistStream**, or **IPersistFile**. Typically, for example, you would implement **IPersistStorage** on an embedded object, **IPersistFile** on a linked object, and **IPersistStream** on a new moniker class, although their uses are not limited to these objects. You could implement **IPersist** in a situation where all that is required is to obtain the CLSID of a persistent object, as it is used in marshaling.

When to Use

The single method of **IPersist** is rarely called directly in application code. It is called by the default object handler to get the CLSID of an embedded object, or an object to be marshaled. A container application, for example, would probably not call the **GetClassID** method directly unless it provided object handlers for specific classes of objects.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IPersist Method	Description
GetClassID	Returns the class identifier (CLSID) for the component object.

IPersist::GetClassID Quick Info

Retrieves the class identifier (CLSID) of an object. The CLSID is a unique value that identifies the code that can manipulate the persistent data.

```
HRESULT GetClassID(
  CLSID *pClassID          //Pointer to CLSID of object
);
```

Parameter

pClassID

[out]Points to the location of the CLSID on return. The CLSID is a globally unique identifier (GUID) that uniquely represents an object class that defines the code that can manipulate the object's data.

Return Values

S_OK

The CLSID was successfully retrieved.

E_FAIL

The CLSID could not be retrieved.

Remarks

The **GetClassID** method retrieves the class identifier (CLSID) for an object, used in later operations to load object-specific code into the caller's context.

Notes to Callers

A container application might call this method to retrieve the original CLSID of an object that it is treating as a different class. Such a call would be necessary if a user performed an editing operation that required the object to be saved. If the container were to save it using the treat-as CLSID, the original application would no longer be able to edit the object. .Typically, in this case, the container calls the **OleSave** helper function, which performs all the necessary steps. For this reason, most container applications have no need to call this method directly.

The exception would be a container that provides an object handler for certain objects. In particular, a container application should not get an object's CLSID and then use it to retrieve class specific information from the registry. Instead, the container should use **IOleObject** and **IDataObject** interfaces to retrieve such class-specific information directly from the object.

Notes to Implementers

Typically, implementations of this method simply supply a constant CLSID for an object. If, however, the object's **TreatAs** registry key has been set by an application that supports emulation (and so is treating the object as one of a different class), a call to **IPersist::GetClassID** must supply the CLSID specified in the **TreatAs** key. For more information on emulation, refer to **CoTreatAsClass**.

When an object is in the running state, the default handler calls an implementation of **IPersist::GetClassID** that delegates the call to the implementation in the object. When the object is not running, the default handler instead calls the **ReadClassStg** function to read the CLSID that is saved in

the object's storage.

If you are writing a custom object handler for your object, you might want to simply delegate this method to the default handler implementation (see **OleCreateDefaultHandler**).

See Also

IPersistStorage, **IPersistStream**, **IPersistFile**, **ReadClassStg**

IPersistFile Quick Info

The **IPersistFile** interface provides methods that permit an object to be loaded from or saved to a disk file, rather than a storage object or stream. Because the information needed to open a file varies greatly from one application to another, the implementation of **IPersistFile::Load** on the object must also open its disk file.

The **IPersistFile** interface inherits its definition from **IPersist**, so all implementations must also include the **GetClassID** method of **IPersist**.

When to Implement

Implement **IPersistFile** when you want to read or write information from a separate file, which could be of any file format.

This interface should be implemented on any objects that support linking through a file moniker, including the following:

- Any object that supports links to its files or to *pseudo-objects* within its files
- A container application that supports links to objects within its compound file

Typically, you implement the **IPersistFile** interface as part of an aggregate object that includes other interfaces that are appropriate for the type of moniker binding that is supported.

For example, in either of the cases mentioned above, the moniker for the linked object can be a composite moniker. In the first case, a composite moniker identifies the pseudo-object contained within the file. In the second case, a composite moniker identifies the embedded object contained within the compound file. In either case of composite monikers, you must implement the **IPersistFile** interface as part of the same object on which the **IOleItemContainer** interface is implemented. Then, when the application for the linked object is run, OLE queries for the **IOleItemContainer** interface to locate the embedded object or the pseudo-object contained in the file.

As another example, if the moniker is a simple file moniker (i.e., the link is to the entire file), OLE queries for the interface that the initiator of the bind operation requested. Typically, this is one of the compound document interfaces, such as **IOleObject**, **IDataObject**, or **IPersistStorage**.

When to Use

Call methods in the **IPersistFile** interface to load or save a linked object in a specified file.

When **IPersistFile** is implemented on an object that supports linking through a file moniker and the application for the linked object is run, OLE calls **IPersistFile::Load**. Once the file is loaded, OLE calls **IPersistFile::QueryInterface** to get another interface pointer to the loaded object. The **IPersistFile** interface is typically part of an aggregate object that offers other interfaces.

In this case, the only **IPersistFile** method that OLE calls is the **Load** method to load a file linked to a container, running the application associated with the file. It would also be unusual for applications to call other methods in this case, which support saving an object to a file. Generally, it is left to the end user and the application for the linked object to decide when to save the object. This differs from the situation for an embedded object, in which the container application uses the **IPersistStorage** interface to provide the storage and to tell the object when to save itself.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported

	interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IPersist Method	Description
GetClassID	Returns the class identifier (CLSID) for the component object.

IPersistFile Methods	Description
IsDirty	Checks an object for changes since it was last saved to its current file.
Load	Opens the specified file and initializes an object from the file contents.
Save	Saves the object into the specified file.
SaveCompleted	Notifies the object that it can revert from NoScribble mode to Normal mode.
GetCurFile	Gets the current name of the file associated with the object.

IPersistFile::GetCurFile Quick Info

Retrieves either the absolute path to the object's current working file or, if there is no current working file, the object's default filename prompt.

```
HRESULT GetCurFile(
  LPOLESTR *ppszFileName          //Pointer to the path for the current file or the default save prompt
);
```

Parameter

ppszFileName

[out]Points to the location of a pointer to a zero-terminated string containing the path for the current file or the default filename prompt (such as **.txt*). If an error occurs, *ppszFileName* is set to NULL.

Return Values

S_OK

A valid absolute path was successfully returned.

S_FALSE

The default save prompt was returned.

E_OUTOFMEMORY

The operation failed due to insufficient memory.

E_FAIL

The operation failed due to some reason other than insufficient memory.

Remarks

This method returns the current filename or the default save prompt for the object.

This method allocates memory for the string returned in the *ppszFileName* parameter using the **IMalloc::Alloc** method. The caller is responsible for calling the **IMalloc::Free** method to free the string. Both the caller and this method use the OLE task allocator provided by a call to **CoGetMalloc**.

The **LPOLESTR** type indicates a wide character string (two bytes per character); otherwise, the string has one byte per character.

The filename returned in *ppszFileName* is the name specified in a call to **IPersistFile::Load** when the document was loaded; or in **IPersistFile::SaveCompleted** if the document was saved to a different file.

If the object does not have a current working file, it should supply the default filename prompt that it would display in a "Save As" dialog. For example, the default save prompt for a word processor object could be:

*.txt

Notes to Callers

OLE does not call the **IPersistFile::GetCurFile** method. Applications would not call this method unless they are also calling the save methods of this interface.

In saving the object, you can call this method before calling **IPersistFile:Save** to determine whether the object has an associated file. If this method returns S_OK, you can then call **IPersistFile::Save** with a NULL filename and a TRUE value for the *fRemember* parameter to tell the object to save itself to its current file. If this method returns S_FALSE, you can use the save prompt returned in the *ppszFileName* parameter to ask the end user to provide a filename. Then, you can call **IPersistFile::Save** with the filename that the user entered to perform a "Save As" operation.

See Also

IPersistFile::Load, **IPersistFile::Save**, **IPersistFile::SaveCompleted**

IPersistFile::IsDirty Quick Info

Checks an object for changes since it was last saved to its current file.

HRESULT IsDirty(*void***);**

Return Values

S_OK

The object has changed since it was last saved.

S_FALSE

The object has not changed since the last save.

Remarks

This method checks whether an object has changed since it was last saved. Call it to determine whether an object should be saved before closing it. The dirty flag for an object is conditionally cleared in the **IPersistFile::Save** method.

Notes to Callers

OLE does not call **IPersistFile::IsDirty**. Applications would not call it unless they are also saving an object to a file.

You should treat any error return codes as an indication that the object has changed. Unless this method explicitly returns S_FALSE, assume that the object must be saved.

Notes to Implementers

An object with no contained objects simply checks its dirty flag to return the appropriate result.

A container with one or more contained objects must maintain an internal dirty flag that is set when any of its contained objects has changed since it was last saved. To do this, the container should maintain an advise sink by implementing the **IAdviseSink** interface. Then, the container can register each link or embedding for data change notifications with a call to **IDataObject::DAdvise**. Then, the container can set its internal dirty flag when it receives an **IAdviseSink::OnDataChange** notification. If the container does not register for data change notifications, the **IPersistFile::IsDirty** implementation would call **IPersistStorage::IsDirty** for each of its contained objects to determine whether they have changed.

The container can clear its dirty flag whenever it is saved, as long as the file to which the object is saved is the current working file after the save. Therefore, the dirty flag would be cleared after a successful "Save" or "Save As" operation, but not after a "Save A Copy As . . ." operation.

See Also

IAdviseSink::OnDataChange, **IDataObject::DAdvise**, **IPersistStorage::IsDirty**

IPersistFile::Load Quick Info

Opens the specified file and initializes an object from the file contents.

```
HRESULT Load(
  LPCOLESTR pszFileName,      //Pointer to absolute path of the file to open
  DWORD dwMode                //Specifies the access mode from the STGM enumeration
);
```

Parameters

pszFileName

[in]Points to a zero-terminated string containing the absolute path of the file to open.

dwMode

[in]Specifies some combination of the values from the **STGM** enumeration to indicate the access mode to use when opening the file. **IPersistFile::Load** can treat this value as a suggestion, adding more restrictive permissions if necessary. If *dwMode* is zero, the implementation should open the file using whatever default permissions are used when a user opens the file.

Return Values

S_OK

The object was successfully loaded.

E_OUTOFMEMORY

The object could not be loaded due to a lack of memory.

E_FAIL

The object could not be loaded for some reason other than a lack of memory.

IPersistFile::Load STG_E_* error codes.

Remarks

IPersistFile::Load loads the object from the specified file. This method is for initialization only and does not show the object to the end user. It is not equivalent to what occurs when an end user selects the File Open command.

Notes to Callers

The **BindToObject** method in file monikers calls this method to load an object during a moniker binding operation (when a linked object is run). Typically, applications do not call this method directly.

Notes to Implementers

Because the information needed to open a file varies greatly from one application to another, the object on which this method is implemented must also open the file specified by the *pszFileName* parameter. This differs from the **IPersistStorage::Load** and **IPersistStream::Load**, in which the caller opens the storage or stream and then passes an open storage or stream pointer to the loaded object.

For an application that normally uses OLE compound files, your **IPersistFile::Load** implementation can simply call the **StgOpenStorage** function to open the storage object in the specified file. Then, you can proceed with normal initialization. Applications that do not use storage objects can perform normal file-opening procedures.

When the object has been loaded, your implementation should register the object in the Running Object Table (see **IRunningObjectTable::Register**).

See Also

IRunningObjectTable::Register, **StgOpenStorage**

IPersistFile::Save Quick Info

Saves a copy of the object into the specified file.

```
HRESULT Save(
  LPCOLESTR pszFileName,      //Pointer to absolute path of the file where the object is saved
  BOOL fRemember              //Specifies whether the file is to be the current working file or not
);
```

Parameters

pszFileName

[in]Points to a zero-terminated string containing the absolute path of the file to which the object should be saved. If *pszFileName* is NULL, the object should save its data to the current file, if there is one.

fRemember

[in]Indicates whether the *pszFileName* parameter is to be used as the current working file. If TRUE, *pszFileName* becomes the current file and the object should clear its dirty flag after the save. If FALSE, this save operation is a "Save A Copy As ..." operation. In this case, the current file is unchanged and the object should not clear its dirty flag. If *pszFileName* is NULL, the implementation should ignore the *fRemember* flag.

Return Values

S_OK

The object was successfully saved.

E_FAIL

The file was not saved.

IPersistFile::Save STG_E_* errors.

Remarks

This method can be called to save an object to the specified file in one of three ways:

Save

Call **IPersistFile::GetCurFile** first to determine whether the object has an associated filename. If so, call **IPersistFile::Save** specifying NULL for the *pszFileName* parameter in this method to indicate that the object should be saved to its current file. Then call **IPersistFile::SaveCompleted** to indicate completion.

Save As

Call **IPersistFile::Save** specifying TRUE in the *fRemember* parameter and a non-NULL value, indicating the name of the new file the object is to be saved to, for the *pszFileName* parameter . Then call **IPersistFile::SaveCompleted** to indicate completion.

Save a Copy As

Call **IPersistFile::Save** specifying FALSE in the *fRemember* parameter and a non-NULL value,

indicating the name of the new file the object is to be copied to, for the *pszFileName* parameter.

The implementer must detect which type of save operation the caller is requesting. If the *pszFileName* parameter is NULL, a "Save" is being requested. If the *pszFileName* parameter is not NULL, use the value of the *fRemember* parameter to distinguish between a "Save As" and a "Save a Copy As".

In "Save" or "Save As" operations, **IPersistFile::Save** clears the internal dirty flag after the save and sends **IAdviseSink::OnSave** notifications to any advisory connections (see also **IOleAdviseHolder::SendOnSave**). Also, in these operations, the object is in NoScribble mode until it receives an **IPersistFile::SaveCompleted** call. In NoScribble mode, the object must not write to the file.

In the "Save As" scenario, the implementation should also send **IAdviseSink::OnRename** notifications to any advisory connections (see also **IOleAdviseHolder::SendOnRename**).

In the "Save a Copy As" scenario, the implementation does not clear the internal dirty flag after the save.

Notes to Callers

OLE does not call **IPersistFile::Save**. Typically, applications would not call it unless they are saving an object to a file directly, which is generally left to the end-user.

See Also

IOleAdviseHolder::SendOnRename, **IOleAdviseHolder::SendOnSave**, **IPersistFile::GetCurFile**, **IPersistFile::SaveCompleted**

IPersistFile::SaveCompleted Quick Info

Notifies the object that it can write to its file. It does this by notifying the object that it can revert from NoScribble mode (in which it must not write to its file), to Normal mode (in which it can). The component enters NoScribble mode when it receives an **IPersistFile::Save** call.

```
HRESULT SaveCompleted(
  LPCOLESTR pszFileName          //Pointer to absolute path of the file where the object was saved
);
```

Parameter

pszFileName

[in]Points to the absolute path of the file where the object was previously saved.

Return Value

S_OK

Returned in all cases.

Remarks

IPersistFile::SaveCompleted is called when a call to **IPersistFile::Save** is completed, and the file that was saved is now the current working file (having been saved with "Save" or "Save As" operations). The call to **Save** puts the object into NoScribble mode so it cannot write to its file. When **SaveCompleted** is called, the object reverts to Normal mode, in which it is free to write to its file.

Notes to Callers

OLE does not call the **IPersistFile::SaveCompleted** method. Typically, applications would not call it unless they are saving objects directly to files, an operation which is generally left to the end-user.

See Also

IPersistFile::Save

IPersistMemory Quick Info

The **IPersistMemory** interface operates exactly as **IPersistStreamInit**, except that it allows the caller to provide a fixed-size memory block (identified with a **void** *) as opposed to **IPersistStreamInit** which involves an arbitrarily expandable **IStream**.

The *cbSize* argument to the **Load** and **Save** methods indicate the amount of memory accessible through *pvMem*.

The **IsDirty**, **GetSizeMax**, and **InitNew** methods are semantically and syntactically identical to those in **IPersistStreamInit**. Only **Load** and **Save** differ.

When to Implement

An object implements this interface to save itself in memory.

When to Use

A container calls the methods of this interface to instruct an object to save and load itself in memory.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPersist Method	Description
GetClassID	Returns the class identifier (CLSID) for the component object.

IPersistMemory Methods	Description
IsDirty	Checks the object for changes since it was last saved.
Load	Initializes an object from the memory block where it was previously saved.
Save	Saves an object into the specified memory block and indicates whether the object should reset its dirty flag.
GetSizeMax	Returns the size in bytes of the memory block needed to save the object.
InitNew	Initializes an object to a default state.

IPersistMemory::GetSizeMax Quick Info

Returns the size in bytes of the memory block needed to save the object.

```
HRESULT GetSizeMax(
  ULARGE_INTEGER* pcbSize      //Pointer to size of memory needed to save object
);
```

Parameter

pcbSize

[out]Pointer to a 64-bit unsigned integer value indicating the size in bytes of the memory needed to save this object.

Return Value

S_OK

The size was successfully returned.

Remarks

This method returns the size needed to save an object. You can call this method to determine the size and set the necessary buffers before calling the **IPersistMemory::Save** method.

Notes to Implementers

The **GetSizeMax** implementation must return a conservative estimate of the necessary size because the **IPersistMemory::Save** method uses a fixed size memory block.

See Also

IPersistMemory::Save

IPersistMemory::InitNew Quick Info

Initializes the object to a default state. This method is called instead of **IPersistMemory::Load**.

HRESULT InitNew(void);

Return Values

S_OK

The object successfully initialized itself.

E_NOTIMPL

The object requires no default initialization. This error code is allowed because an object may choose to implement **IPersistMemory** simply for orthogonality or in anticipation of a future need for this method.

E_UNEXPECTED

This method was called after the object was already initialized with **IPersistMemory::Load**. Only one initialization is allowed per instance.

E_OUTOFMEMORY

There was not enough memory for the object to initialize itself.

Notes to Implementers

If the object has already been initialized with **Load**, then this method must return E_UNEXPECTED.

See Also

IPersistMemory::Load

IPersistMemory::IsDirty Quick Info

Checks the object for changes since it was last saved.

HRESULT IsDirty(void);

Return Values

S_OK

The object has changed since it was last saved.

S_FALSE

The object has not changed since the last save.

Remarks

This method checks whether an object has changed since it was last saved so you can avoid losing information in objects that have not yet been saved. The dirty flag for an object is conditionally cleared in the **IPersistMemory::Save** method.

Notes to Callers

You should treat any error return codes as an indication that the object has changed. In other words, unless this method explicitly returns S_FALSE, you must assume that the object needs to be saved.

See Also

IPersistMemory::Save

IPersistMemory::Load Quick Info

Instructs the object to load its persistent data from the memory pointed to by *pvMem* where *cbSize* indicates the amount of memory at *pvMem*. The object must not read past the address *(BYTE*)((BYTE *)pvMem+cbSize)*.

```
HRESULT Load(
  void* pvMem,      //Pointer to the stream from which the object should be loaded
  ULONG cbSize      //Amount of memory from which the object can read its data
);
```

Parameters

pvMem

[in] Pointer to the address in memory from which the object can read up to *cbSize* bytes of its data.

cbSize

[in] The amount of memory available at *pvMem* from which the object can read its data.

Return Values

S_OK

The object successfully loaded its data.

E_UNEXPECTED

This method was called after the object was already initialized with **IPersistMemory::Load**. Only one initialization is allowed per instance.

E_POINTER

The pointer in *pvMem* is NULL.

Remarks

Any object that implements **IPersistMemory** has some information to load persistently, therefore E_NOTIMPL is not a valid return code.

See Also

IPersistMemory::InitNew

IPersistMemory::Save

Quick Info

Instructs the object to save its persistent data to the memory pointed to by *pvMem* where *cbSize* indicates the amount of memory available at *pvMem*. The object must not write past the address *(BYTE*)((BYTE *)pvMem+cbSize)*. The *fClearDirty* flag determines whether the object is to clear its dirty state after the save is complete.

```
HRESULT Save(
  void* pvMem,          //Pointer to the stream where the object is to be saved
  BOOL fClearDirty,     //Specifies whether to clear the dirty flag
  ULONG cbSize          //Amount of memory to which the object can write its data
);
```

Parameters

pvMem

[in] Pointer to the memory in which the object should save up to *cbSize* bytes of its data.

fClearDirty

[in] A flag indicating whether the object should clear its dirty state on return from **Save** or leave that state as-is.

cbSize

[in] The amount of memory available at *pvMem* to which the object can write its data.

Return Values

S_OK

The object successfully initialized itself.

E_UNEXPECTED

This method was called before the object was initialized with **IPersistMemory::InitNew** or **IPersistMemory::Load**.

E_INVALIDARG

The number of bytes indicated by *cbSize* is too small to allow the object to save itself completely.

E_POINTER

The pointer in *pvMem* is NULL.

Remarks

Any object that implements **IPersistMemory** has some information to save persistently, therefore E_NOTIMPL is not a valid return code.

The caller should ideally allocate as many bytes as the object returns from **IPersistMemory::GetSizeMax**.

See Also

IPersistMemory::InitNew, **IPersistMemory::Load**

IPersistPropertyBag Quick Info

The **IPersistPropertyBag** interface works in conjunction with **IPropertyBag** and **IErrorLog** to define an individual property-based persistence mechanism. Whereas a mechanism like **IPersistStream** gives an object an **IStream** in which to store its binary data, **IPersistPropertyBag** provides an object with an **IPropertyBag** interface through which it can save and load individual properties. The implementer of **IPropertyBag** can then save those properties in whatever way it chooses, such as name/value pairs in a text file. Errors encountered in the process (on either side) are recorded in an error log through **IErrorLog**. This error reporting mechanism work on a per-property basis instead of an all properties as a whole basis through just the return value of **IPersist*::Load** or **IPersist*::Save**.

The basic mechanism is that a container tells the object to save or load its properties through **IPersistPropertyBag**. For each property, the object calls the container's **IPropertyBag** interface passed to the **IPersistPropertyBag** methods. **IPropertyBag::Write** saves a property in whatever place the container wants to put it, and **IPropertyBag::Read** retrieves a property.

This protocol is essentially a means of sequentially communicating individual property values from the object to the container, which is useful for doing save-as-text operations and the like. The object gives the container the choice of the format in which each property is saved, while retaining itself the decision as to which properties are saved or loaded.

When to Implement

An object implements this interface to enable saving its properties persistently.

When to Use

A container calls the methods on this interface to instruct an object to load and save its properties to the supplied property bag.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPersist Method	Description
GetClassID	Returns the class identifier (CLSID) for the component object.

IPersistPropertyBag Methods	Description
InitNew	Called by the container when the control is initialized to initialize the property bag.
Load	Called by the container to load the control's properties.
Save	Called by the container to save the object's properties.

See Also

IErrorLog, **IPropertyBag**

IPersistPropertyBag::InitNew

Called by the container when the control is initialized to initialize the property bag.

HRESULT InitNew(void);

Return Values

S_OK

The object successfully initialized itself. This should be returned even if the object doesn't do anything in the method.

CO_E_ALREADYINITIALISED

The object has already been initialized.

E_OUTOFMEMORY

The storage object was not initialized due to a lack of memory.

E_UNEXPECTED

The storage object was not initialized due to some reason other than a lack of memory.

Remarks

This method informs the object that it is being initialized as a newly created object.

E_NOTIMPL should not be returned—use S_OK when the object has nothing to do in the method.

See Also

IPersistPropertyBag::Load

IPersistPropertyBag::Load

Called by the container to load the control's properties.

```
HRESULT Load(

  IPropertyBag* pPropBag,     //Pointer to caller's property bag
  IErrorLog* pErrorLog        //Pointer to error log
);
```

Parameters

pPropBag

[in] Pointer to the caller's **IPropertyBag** interface bag that the control uses to read its properties. Cannot be NULL.

pErrorLog

[in] Pointer to the caller's **IErrorLog** interface in which the object stores any errors that occur during initialization. Can be NULL in which case the caller is not interested in errors.

Return Values

S_OK

The object successfully initialized itself.

E_UNEXPECTED

This method was called after **IPersistPropertyBag::InitNew** has already been called. They two initialization methods are mutually exclusive.

E_OUTOFMEMORY

The properties were not loaded due to a lack of memory.

E_POINTER

The address in *pPropBag* is not valid (such as NULL) and therefore the object cannot initialize itself.

E_FAIL

The object was unable to retrieve a critical property that is necessary for the object's successful operation. The object was therefore unable to initialize itself completely.

Remarks

This method instructs the object to initialize itself using the properties available in the property bag, notifying the provided error log object when errors occur. All property storage must take place within this method call as the object cannot hold the **IPropertyBag** pointer.

E_NOTIMPL is not a valid return code as any object implementing this interface must support the entire functionality of the interface.

See Also

IPersistPropertyBag::InitNew

IPersistPropertyBag::Save

Called by the container to save the object's properties.

```
HRESULT Save(
  IPropertyBag* pPropBag,      //Pointer to the caller's property bag
  BOOL fClearDirty,            //Specifies whether to clear the dirty flag
  BOOL fSaveAllProperties      //Specifies whether to save all properties or just those that have changed
);
```

Parameters

pPropBag

[in] Pointer to the caller's **IPropertyBag** interface through which the object can write properties. Cannot be NULL.

fClearDirty

[in] A flag indicating whether the object should clear its dirty flag when saving is complete. TRUE means clear the flag, FALSE means leave the flag unaffected. FALSE is used when the caller wishes to do a Save Copy As type of operation.

fSaveAllProperties

[in] A flag indicating whether the object should save all its properties (TRUE) or only those that have changed since the last save or initialization (FALSE).

Return Values

S_OK

The object successfully saved the requested properties itself.

E_FAIL

There was a problem saving one of the properties. The object can choose to fail only if a necessary property could not be saved, meaning that the object can assume default property values if a given property is not seen through **IPersistPropertyBag::Load** at some later time.

E_POINTER

The address in *pPropBag* is not valid (such as NULL) and therefore the object cannot initialize itself.

STG_E_MEDIUMFULL

The object was not saved because of a lack of space on the disk.

Remarks

This method instructs the object to save its properties to the specified property bag, optionally clearing the object's dirty flag. The caller can request that the object save all properties or that the object save only those that are known to have changed.

E_NOTIMPL is not a valid return code as any object implementing this interface must support the entire functionality of the interface.

See Also

IPersistPropertyBag::InitNew, **IPersistPropertyBag::Load**

IPersistStorage

Quick Info

The **IPersistStorage** interface defines methods that enable a container application to pass a storage object to one of its contained objects and to load and save the storage object. This interface supports the structured storage model, in which each contained object has its own storage that is nested within the container's storage.

The **IPersistStorage** contract inherits its definition from **IPersist**, so all implementations must also include the **GetClassID** method of **IPersist**.

When to Implement

Any object that can be embedded in a container must implement the **IPersistStorage** interface. This interface is one of the primary interfaces for a compound document object. Embeddable objects must also implement the **IOleObject** and **IDataObject** interfaces.

The OLE default handler for embedded objects provides an implementation of the **IPersistStorage** interface that is used when the object is in the loaded state. Similarly, the OLE default link handler provides an **IPersistStorage** implementation that manages storage for a linked object. These default handlers both interact with the OLE default cache implementation, which has its own **IPersistStorage** implementation.

If you are providing a custom embedding or link handler for your objects, the handler must include an implementation of **IPersistStorage**. You can delegate calls to the default handler so you can take advantage of the default cache implementation.

When to Use

When an OLE container creates a new object, loads an existing object from storage, or inserts a new object in a clipboard or a drag-and-drop operation, the container uses the **IPersistStorage** interface to initialize the object and put it in the loaded or running state. When an object is loaded or running, an OLE container calls other **IPersistStorage** methods to instruct the object to perform various save operations or to release its storage.

Typically, applications use helper functions such as **OleLoad** or **OleCreate**, rather than calling the **IPersistStorage::Load** or **IPersistStorage::InitNew** methods directly. Similarly, applications typically call the **OleSave** helper function rather than calling **IPersistStorage::Save** directly.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPersist Method	**Description**
GetClassID	Returns the class identifier (CLSID) for the object on which it is implemented.

IPersistStorage Methods	**Description**
IsDirty	Indicates whether the object has changed since it was last saved to

	its current storage.
InitNew	Initializes a new storage object.
Load	Initializes an object from its existing storage.
Save	Saves an object, and any nested objects that it contains, into the specified storage object. The object enters NoScribble mode.
SaveCompleted	Notifies the object that it can revert from NoScribble or HandsOff mode, in which in must not write to its storage object, to Normal mode in which it can.
HandsOffStorage	Instructs the object to release all storage objects that have been passed to it by its container and to enter HandsOffAfterSave or HandsOffFromNormal mode.

See Also

IDataObject, **IOleObject**, **OleCreate**, **OleLoad**, **OleSave**

IPersistStorage::HandsOffStorage Quick Info

Instructs the object to release all storage objects that have been passed to it by its container and to enter HandsOff mode, in which the object cannot do anything and the only operation that works is a close operation.

HRESULT HandsOffStorage(*void***);**

Return Value

S_OK

The object has successfully entered HandsOff mode.

Remarks

This method causes an object to release any storage objects that it is holding and to enter the HandsOff mode until a subsequent **IPersistStorage::SaveCompleted** call. In HandsOff mode, the object cannot do anything and the only operation that works is a close operation.

A container application typically calls this method during a full save or low-memory full save operation to force the object to release all pointers to its current storage. In these scenarios, the **HandsOffStorage** call comes after a call to either **OleSave** or **IPersistStorage::Save**, putting the object in HandsOffAfterSave mode. Calling this method is necessary so the container application can delete the current file as part of a full save, or so it can call the **IRootStorage::SwitchToFile** method as part of a low-memory save.

A container application also calls this method when an object is in Normal mode to put the object in HandsOffFromNormal mode.

While the component object is in either HandsOffAfterSave or HandsOffFromNormal mode, most operations on the object will fail. Thus, the container should restore the object to Normal mode as soon as possible. The container application does this by calling the **IPersistStorage::SaveCompleted** method, which passes a storage pointer back to the component object for the new storage object.

Notes to Implementers

This method must release all pointers to the current storage object, including pointers to any nested streams and storages. If the object contains nested objects, the container application must recursively call this method for any nested objects that are loaded or running.

See Also

OleSave, **IPersistStorage::Save**, **IPersistStorage::SaveCompleted**, **IRootStorage::SwitchToFile**

IPersistStorage::InitNew Quick Info

Initializes a new object, providing a pointer to the storage to be used for the object.

```
HRESULT InitNew(
  IStorage *pStg          //Points to the new storage object
);
```

Parameter

pStg

[in]**IStorage** pointer to the new storage object to be initialized. The container creates a nested storage object in its storage object (see **IStorage::CreateStorage**). Then, the container calls the **WriteClassStg** function to initialize the new storage object with the object class identifier (CLSID).

Return Values

S_OK

The new storage object was successfully initialized.

CO_E_ALREADYINITIALIZED

The object has already been initialized by a previous call to either the **IPersistStorage::Load** method or the **IPersistStorage::InitNew** method.

E_OUTOFMEMORY

The storage object was not initialized due to a lack of memory.

E_FAIL

The storage object was not initialized for some reason other than a lack of memory.

Remarks

A container application can call this method when it needs to initialize a new object, for example, with an InsertObject command.

An object that supports the **IPersistStorage** interface must have access to a valid storage object at all times while it is running. This includes the time just after the object has been created but before it has been made persistent. The object's container must provide the object with a valid **IStorage** pointer to the storage during this time through the call to **IPersistStorage::InitNew**. Depending on the container's state, a temporary file might have to be created for this purpose.

If the object wants to retain the **IStorage** instance, it must call **IUnknown::AddRef** to increment its reference count.

After the call to **IPersistStorage::InitNew**, the object is in either the loaded or running state. For example, if the object class has an in-process server, the object will be in the running state. However, if the object uses the default handler, the container's call to **InitNew** only invokes the handler's implementation which does not run the object. Later if the container runs the object, the handler calls the **IPersistStorage::InitNew** method for the object.

Notes to Callers

Rather than calling **IPersistStorage::InitNew** directly, you typically call the **OleCreate** helper function which does the following:

1. Calls the **CoCreateInstance** function to create an instance of the object class
2. Queries the new instance for the **IPersistStorage** interface
3. Calls the **IPersistStorage::InitNew** method to initialize the object

The container application should cache the **IPersistStorage** pointer to the object for use in later operations on the object.

Notes to Implementers

An implementation of **IPersistStorage::InitNew** should initialize the object to its default state, taking the following steps:

1. Pre-open and cache the pointers to any streams or storages that the object will need to save itself to this storage.
2. Call **IPersistStorage::AddRef** and cache the storage pointer that is passed in.
3. Call the **WriteFmtUserTypeStg** function to write the native clipboard format and user type string for the object to the storage object.
4. Set the dirty flag for the object.

The first two steps are particularly important for ensuring that the object can save itself in low memory situations. Pre-opening and holding onto pointers to the stream and storage interfaces guarantee that a save operation to this storage will not fail due to insufficient memory.

Your implementation of this method should return the CO_E_ALREADYINITIALIZED error code if it receives a call to either the **IPersistStorage::InitNew** method or the **IPersistStorage::Load** method after it is already initialized.

See Also

IPersistStorage::Load, **OleCreate**, **WriteFmtUserTypeStg**

IPersistStorage::IsDirty Quick Info

Indicates whether the object has changed since it was last saved to its current storage.

HRESULT IsDirty(*void***);**

Return Values

S_OK

The object has changed since it was last saved.

S_FALSE

The object has not changed since the last save.

Remarks

This method checks whether an object has changed since it was last saved so you can save it before closing it. The dirty flag for an object is conditionally cleared in the **IPersistStorage::Save** method.

For example, you could optimize a File:Save operation by calling the **IPersistStorage::IsDirty** method for each object and then calling the **IPersistStorage::Save** method only for those objects that are dirty.

Notes to Callers

You should treat any error return codes as an indication that the object has changed. In other words, unless this method explicitly returns S_FALSE, you must assume that the object needs to be saved.

Notes to Implementers

A container with one or more contained objects must maintain an internal dirty flag that is set whenever any of its contained objects are dirty.

See Also

IAdviseSink::OnDataChange, **IDataObject::DAdvise**, **IPersistStorage::Save**

IPersistStorage::Load Quick Info

Loads an object from its existing storage.

```
HRESULT Load(
  IStorage *pStg          //Pointer to existing storage for the object
);
```

Parameter

pStg

[in]**IStorage** pointer to the existing storage from which the object is to be loaded.

Return Values

S_OK

The object was successfully loaded.

CO_E_ALREADYINITIALIZED

The object has already been initialized by a previous call to the **IPersistStorage::Load** method or the **IPersistStorage::InitNew** method.

E_OUTOFMEMORY

The object was not loaded due to lack of memory.

E_FAIL

The object was not loaded due to some reason besides a lack of memory.

Remarks

This method initializes an object from an existing storage. The object is placed in the loaded state if this method is called by the container application. If called by the default handler, this method places the object in the running state.

Either the default handler or the object itself can hold onto the **IStorage** pointer while the object is loaded or running.

Notes to Callers

Rather than calling **IPersistStorage::Load** directly, you typically call the **OleLoad** helper function which does the following:

1. Create an uninitialized instance of the object class
2. Query the new instance for the **IPersistStorage** interface
3. Call **IPersistStorage::Load** to initialize the object from the existing storage

You also call this method indirectly when you call the **OleCreateFromData** function or the **OleCreateFromFile** function to insert an object into a compound file (as in a drag-and-drop or clipboard paste operation).

The container should cache the **IPersistStorage** pointer for use in later operations on the object.

Notes to Implementers

Your implementation should perform the following steps to load an object:

1. Open the object's streams in the storage object, and read the necessary data into the object's internal data structures.
2. Clear the object's dirty flag.
3. Call the **IPersistStorage::AddRef** method and cache the passed in storage pointer.
4. Keep open and cache the pointers to any streams or storages that the object will need to save itself to this storage.
5. Perform any other default initialization required for the object.

Steps 3 and 4 are particularly important for ensuring that the object can save itself in low memory situations. Holding onto pointers to the storage and stream interfaces guarantees that a save operation to this storage will not fail due to insufficient memory.

Your implementation of this method should return the CO_E_ALREADYINITIALIZED error code if it receives a call to either the **IPersistStorage::InitNew** method or the **IPersistStorage::Load** method after it is already initialized.

See Also

GetConvertStg, **IPersistStorage::InitNew**, **OleLoad**, **ReadFmtUserTypeStg**, **SetConvertStg**, **WriteFmtUserTypeStg**

IPersistStorage::Save Quick Info

Saves an object, and any nested objects that it contains, into the specified storage. The object is placed in NoScribble mode, and it must not write to the specified storage until it receives a call to its **IPersistStorage::SaveCompleted** method.

```
HRESULT Save(
  IStorage *pStgSave,          //Pointer to storage object
  BOOL fSameAsLoad             //Indicates whether the specified storage object is the current one
);
```

Parameters

pStgSave

[in]**IStorage** pointer to the storage into which the object is to be saved.

fSameAsLoad

[in]Indicates whether the specified storage is the current one, which was passed to the object by one of the following calls:

- **IPersistStorage::InitNew** when it was created.
- **IPersistStorage::Load** when it was loaded.
- **IPersistStorage::SaveCompleted** when it was saved to a storage different from its current storage.

This parameter is set to FALSE when performing a Save As or Save A Copy To operation or when performing a full save. In the latter case, this method saves to a temporary file, deletes the original file, and renames the temporary file.

This parameter is set to TRUE to perform a full save in a low-memory situation or to perform a fast incremental save in which only the dirty components are saved.

Return Values

S_OK

The object was successfully saved.

STG_E_MEDIUMFULL

The object was not saved because of a lack of space on the disk.

E_FAIL

The object could not be saved due to errors other than a lack of disk space.

Remarks

This method saves an object, and any nested objects it contains, into the specified storage. It also places the object into NoScribble mode. Thus, the object cannot write to its storage until a subsequent call to the **IPersistStorage::SaveCompleted** method returns the object to Normal mode.

If the storage object is the same as the one it was loaded or created from, the save operation may be able to write incremental changes to the storage object. Otherwise, a full save must be done.

This method recursively calls the **IPersistStorage::Save** method, the **OleSave** function, or the **IStorage::CopyTo** method to save its nested objects.

This method does not call the **IStorage::Commit** method. Nor does it write the CLSID to the storage object. Both of these tasks are the responsibilities of the caller.

Notes to Callers

Rather than calling **IPersistStorage::Save** directly, you typically call the **OleSave** helper function which performs the following steps:

1. Call the **WriteClassStg** function to write the class identifier for the object to the storage.
2. Call the **IPersistStorage::Save** method.
3. If needed, call the **IStorage::Commit** method on the storage object.

Then, a container application performs any other operations necessary to complete the save and calls the **SaveCompleted** method for each object.

If an embedded object passes the **IPersistStorage::Save** method to its nested objects, it must receive a call to its **IPersistStorage::SaveCompleted** method before calling this method for its nested objects.

See Also

IPersistStorage::InitNew, **IPersistStorage::Load**, **IPersistStorage::SaveCompleted**, **OleSave**, **IStorage::Commit**, **IStorage::CopyTo**, **OleSave**, **WriteClassStg**, **WriteFmtUserTypeStg**

IPersistStorage::SaveCompleted Quick Info

Notifies the object that it can revert from NoScribble or HandsOff mode, in which it must not write to its storage object, to Normal mode, in which it can. The object enters NoScribble mode when it receives an **IPersistStorage::Save** call.

```
HRESULT SaveCompleted(
  IStorage *pStgNew          //Pointer to the current storage object
);
```

Parameter

pStgNew

[in]**IStorage** pointer to the new storage object, if different from the storage object prior to saving. This pointer can be NULL if the current storage object does not change during the save operation. If the object is in HandsOff mode, this parameter must be non-NULL.

Return Values

S_OK

The object was successfully returned to Normal mode.

E_OUTOFMEMORY

The object remained in HandsOff mode or NoScribble mode due to a lack of memory. Typically, this error occurs when the object is not able to open the necessary streams and storage objects in *pStgNew*.

E_INVALIDARG

The *pStgNew* parameter is not valid. Typically, this error occurs if *pStgNew* is NULL when the object is in HandsOff mode.

E_UNEXPECTED

The object is in Normal mode, and there was no previous call to **IPersistStorage::Save** or **IPersistStorage::HandsOffStorage**.

Remarks

This method notifies an object that it can revert to Normal mode and can once again write to its storage object. The object exits NoScribble mode or HandsOff mode.

If the object is reverting from HandsOff mode, the *pStgNew* parameter must be non-NULL. In HandsOffFromNormal mode, this parameter is the new storage object that replaces the one that was revoked by the **IPersistStorage::HandsOffStorage** method. The data in the storage object is a copy of the data from the revoked storage object. In HandsOffAfterSave mode, the data is the same as the data that was most recently saved. It is not the same as the data in the revoked storage object.

If the object is reverting from NoScribble mode, the *pStgNew* parameter can be NULL or non-NULL. If NULL, the object once again has access to its storage object. If it is not NULL, the component object should simulate receiving a call to its **IPersistStorage::HandsOffStorage** method. If the component object cannot simulate this call, its container must be prepared to actually call the **IPersistStorage::HandsOffStorage** method.

The **IPersistStorage::SaveCompleted** method must recursively call any nested objects that are loaded or running.

If this method returns an error code, the object is not returned to Normal mode. Thus, the container object can attempt different save strategies.

See Also

IAdviseSink::OnSave, **IOleObject::Close**, **IPersistStorage::HandsOffStorage**, **IPersistStorage::Save**, **IRootStorage::SwitchToFile**

IPersistStream

Quick Info

The **IPersistStream** interface provides methods for saving and loading objects that use a simple serial stream for their storage needs. The **IPersistStream** interface inherits its definition from the **IPersist** interface, and so the includes the **GetClassID** method of **IPersist**.

One way in which it is used is to support OLE moniker implementations. Each of the OLE-provided moniker interfaces provides an **IPersistStream** implementation through which the moniker saves or loads itself. An instance of the OLE generic composite moniker class calls the **IPersistStream** methods of its component monikers to load or save the components in the proper sequence in a single stream.

OLE containers with embedded and linked component objects do not use this interface; they use the **IPersistStorage** interface instead.

When to Implement

Implement the **IPersistStream** interface on objects that are to be saved to a simple stream. Some objects of this type are monikers and some OLE controls, although generally, controls use the **IPersistStreamInit** interface, which has the same methods as IPersistStream, with one added method, **IPersistStreamInit::InitNew**. The **IMoniker** interface is derived from the **IPersistStream** interface, so you must implement the **IPersistStream** interface if you are implementing a new moniker class.

When to Use

Call methods of **IPersistStream** from a container application to save or load objects that are contained in a simple stream. When used to save or load monikers, typical applications do not call the methods directly, but allow the default link handler to make the calls to save and load the monikers that identify the link source. These monikers are stored in a stream in the storage for the linked object. If you are writing a custom link handler for your class of objects, you would call the methods of **IPersistStream** to implement the link handler.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IPersist Method	Description
GetClassID	Returns the class identifier (CLSID) for the component object.

IPersistStream Methods	Description
IsDirty	Checks the object for changes since it was last saved.
Load	Initializes an object from the stream where it was previously saved.
Save	Saves an object into the specified stream and indicates whether the object should reset its dirty flag.
GetSizeMax	Return the size in bytes of the stream needed to save the object.

See Also

IMoniker

IPersistStream::GetSizeMax Quick Info

Returns the size in bytes of the stream needed to save the object.

```
HRESULT GetSizeMax(
  ULARGE_INTEGER *pcbSize          //Pointer to size of stream needed to save object
);
```

Parameter

pcbSize

[out]Points to a 64-bit unsigned integer value indicating the size in bytes of the stream needed to save this object.

Return Value

S_OK

The size was successfully returned.

Remarks

This method returns the size needed to save an object. You can call this method to determine the size and set the necessary buffers before calling the **IPersistStream::Save** method.

Notes to Implementers

The **GetSizeMax** implementation should return a conservative estimate of the necessary size because the caller might call the **IPersistStream::Save** method with a non-growable stream.

See Also

IPersistStream::Save

IPersistStream::IsDirty Quick Info

Checks the object for changes since it was last saved.

HRESULT IsDirty(*void***);**

Return Values

S_OK

The object has changed since it was last saved.

S_FALSE

The object has not changed since the last save.

Remarks

This method checks whether an object has changed since it was last saved so you can avoid losing information in objects that have not yet been saved. The dirty flag for an object is conditionally cleared in the **IPersistStream::Save** method.

Notes to Callers

You should treat any error return codes as an indication that the object has changed. In other words, unless this method explicitly returns S_FALSE, you must assume that the object needs to be saved.

Note that the OLE-provided implementations of the **IPersistStream::IsDirty** method in the OLE-provided moniker interfaces always return S_FALSE because their internal state never changes.

See Also

IPersistStream::Save

IPersistStream::Load Quick Info

Initializes an object from the stream where it was previously saved.

```
HRESULT Load(
  IStream *pStm          //Pointer to the stream from which the object should be loaded
);
```

Parameter

pStm

[in]**IStream** pointer to the stream from which the object should be loaded.

Return Values

S_OK

The object was successfully loaded.

E_OUTOFMEMORY

The object was not loaded due to a lack of memory.

E_FAIL

The object was not loaded due to some reason other than a lack of memory.

Remarks

This method loads an object from its associated stream. The seek pointer is set as it was in the most recent **IPersistStream::Save** method. This method can seek and read from the stream, but cannot write to it.

On exit, the seek pointer must be in the same position it was in on entry, immediately past the end of the data.

Notes to Callers

Rather than calling **IPersistStream::Load** directly, you typically call the **OleLoadFromStream** function does the following:

1. Calls the **ReadClassStm** function to get the class identifier from the stream.
2. Calls the **CoCreateInstance** function to create an instance of the object.
3. Queries the instance for **IPersistStream**.
4. Calls **IPersistStream::Load**.

The **OleLoadFromStream** function assumes that objects are stored in the stream with a class identifier followed by the object data. This storage pattern is used by the generic, composite-moniker implementation provided by OLE.

If the objects are not stored using this pattern, you must call the methods separately yourself.

See Also

CoCreateInstance, **OleLoadFromStream**, **ReadClassStm**

IPersistStream::Save Quick Info

Saves an object to the specified stream.

```
HRESULT Save(
  IStream *pStm,          //Pointer to the stream where the object is to be saved
  BOOL fClearDirty        //Specifies whether to clear the dirty flag
);
```

Parameters

pStm

[in]**IStream** pointer to the stream into which the object should be saved.

fClearDirty

[in]Indicates whether to clear the dirty flag after the save is complete. If TRUE, the flag should be cleared. If FALSE, the flag should be left unchanged.

Return Values

S_OK

The object was successfully saved to the stream.

STG_E_CANTSAVE

The object could not save itself to the stream. This error could indicate, for example, that the object contains another object that is not serializable to a stream or that an **IStream::Write** call returned STG_E_CANTSAVE.

STG_E_MEDIUMFULL

The object could not be saved because there is no space left on the storage device.

Remarks

IPersistStream::Save saves an object into the specified stream and indicates whether the object should reset its dirty flag.

The seek pointer is positioned at the location in the stream at which the object should begin writing its data. The object calls the **IStream::Write** method to write its data.

On exit, the seek pointer must be positioned immediately past the object data. The position of the seek pointer is undefined if an error returns.

Notes to Callers

Rather than calling **IPersistStream::Save** directly, you typically call the **OleSaveToStream** helper function which does the following:

1. Calls **IPersistStream::GetClassID** to get the object's CLSID.
2. Calls the **WriteClassStm** function to write the object's CLSID to the stream.
3. Calls **IPersistStream::Save**.

If you call these methods directly, you can write other data into the stream after the CLSID before calling **IPersistStream::Save**.

The OLE-provided implementation of **IPersistStream** follows this same pattern.

Notes to Implementers

The **IPersistStream::Save** method does not write the CLSID to the stream. The caller is responsible for writing the CLSID.

The **IPersistStream::Save** method can read from, write to, and seek in the stream; but it must not seek to a location in the stream before that of the seek pointer on entry.

See Also

IPersist::GetClassID, **IStream::Write**, **OleSaveToStream**

IPersistStreamInit

Quick Info

The **IPersistStreamInit** interface is defined as a replacement for **IPersistStream** in order to add an initialization method, **InitNew**. This interface is not derived from **IPersistStream**; it is mutually exclusive with **IPersistStream**. An object chooses to support only one of the two interfaces, based on whether it requires the **InitNew** method. Otherwise, the signatures and semantics of the other methods are the same as the corresponding methods of **IPersistStream**, except as described below.

When to Implement

Implement this interface on any object that needs to support initialized stream-based persistence, regardless of whatever else the object does. The presence of the **InitNew** method requires some changes to other methods that are common to **IPersistStream**, as noted in the method descriptions.

When to Use

Use this interface to initialize a stream-based object and to save that object to a stream.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPersistStreamInit Methods	Description
IsDirty	Checks the object for changes since it was last saved.
Load	Initializes an object from the stream where it was previously saved.
Save	Saves an object into the specified stream and indicates whether the object should reset its dirty flag.
GetSizeMax	Return the size in bytes of the stream needed to save the object.
InitNew	Initializes an object to a default state.

See Also

IPersistStream

IPersistStreamInit::GetSizeMax Quick Info

Same as **IPersistStream::GetSizeMax**.

```
HRESULT GetSizeMax(
  ULARGE_INTEGER* pcbSize      //Receives a pointer to the size of the stream needed to save object
);
```

IPersistStreamInit::InitNew Quick Info

Initializes the object to a default state. This method is called instead of **IPersistStreamInit::Load**.

HRESULT InitNew(void);

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The object successfully initialized itself.

E_NOTIMPL

The object requires no default initialization. This error code is allowed because an object may choose to implement **IPersistStreamInit** simply for orthogonality or in anticipation of a future need for this method.

Remarks

Notes to Implementers

If the object has already been initialized with **Load**, then this method must return E_UNEXPECTED.

See Also

IPersistStreamInit::Load

IPersistStreamInit::IsDirty Quick Info

Same as **IPersistStream::IsDirty**.

HRESULT IsDirty(void);

IPersistStreamInit::Load Quick Info

Same as **IPersistStream::Load**.

```
HRESULT Load(
  LPSTREAM pStm      //Pointer to the stream from which the object should be loaded
);
```

Remarks

Notes to Implementers

If the object has already been initialized with **InitNew**, then this method must return E_UNEXPECTED.

IPersistStreamInit::Save Quick Info

Same as **IPersistStream::Save**.

```
HRESULT Save(
  LPSTREAM pStm ,      //Pointer to the stream where the object is to be saved
  BOOL fClearDirty     //Specifies whether to clear the dirty flag
);
```

IPicture Quick Info

The **IPicture** interface manages a picture object and its properties. Picture objects provide a language-neutral abstraction for bitmaps, icons, and metafiles. As with the standard font object, the system provides a standard implementation of the picture object. Its primary interfaces are **IPicture** and **IPictureDisp**, the latter being derived from **IDispatch** to provide access to the picture's properties through Automation. A picture object is created with **OleCreatePictureIndirect**.

The picture object also supports the outgoing interface **IPropertyNotifySink**, so a client can determine when picture properties change. Since the picture object supports at least one outgoing interface, it also implements **IConnectionPointContainer** and its associated interfaces for this purpose.

The picture object also supports **IPersistStream** so it can save and load itself from an instance of **IStream**. An object that uses a picture object internally would normally save and load the picture as part of the object's own persistence handling. The function **OleLoadPicture** simplifies the creation of a picture object based on stream contents.

When to Implement

Typically, you use the OLE-provided picture object, which provides the **IPicture** and **IPictureDisp** interfaces for you. The **IPicture** interface is the primary interface implemented by the OLE-provided picture object. It allows the caller to manage picture properties and to use that picture in graphical rendering. Each property in the **IPicture** interface includes a get_*PropertyName* method if the property supports read access and a put_*PropertyName* method if the property supports write access. Most of the properties support only read access with the exception of hPal.

Property	Type	Access Allowed	Description
HANDLE	OLE_HANDLE (int)	R	The Windows GDI handle of the picture
hPal	OLE_HANDLE (int)	RW	The Windows handle of the palette used by the picture
Type	short	R	The type of picture (see below)
Width	OLE_XSIZE_HIMETRIC (long)	R	The width of the picture
Height	OLE_YSIZE_HIMETRIC (long)	R	The height of the picture
CurDC	HDC	R	The current device context
KeepOriginalFormat	BOOL	RW	If TRUE, the picture object maintains the entire original state of the picture in memory. If FALSE, any state not applicable to the user's machine is discarded
Attributes	DWORD	R	Miscellaneous bit attributes of the picture (see below)

When to Use

Use this interface to change the properties of a picture object.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPicture Methods	Description
get_Handle	Returns the Windows GDI handle of the picture managed within this picture object.
get_Hpal	Returns a copy of the palette currently used by the picture object.
get_Type	Returns the current type of the picture.
get_Width	Returns the current width of the picture in the picture object.
get_Height	Returns the current height of the picture in the picture object.
Render	Draws the specified portion of the picture onto the specified device context, positioned at the specified location.
set_Hpal	Sets the current palette of the picture.
get_CurDC	Returns the current device context into which this picture is selected.
SelectPicture	Selects a bitmap picture into a given device context, returning the device context in which the picture was previously selected as well as the picture's GDI handle.
get_KeepOriginalFormat	Returns the current value of the picture object's KeepOriginalFormat property.
put_KeepOriginalFormat	Sets the picture object's KeepOriginalFormat property.
PictureChanged	Notifies the picture object that its picture resource changed.
SaveAsFile	Saves the picture's data into a stream in the same format that it would save itself into a file.
get_Attributes	Returns the current set of the picture's bit attributes.

See Also

IPicture - Ole Implementation, IPictureDisp

IPicture::get_Attributes Quick Info

Returns the current set of the picture's bit attributes.

```
HRESULT get_Attributes(
  DWORD* pdwAttr      //Receives a pointer to attribute value.
);
```

Parameters

pdwAttr

[out] Pointer to the caller's variable that receives the attribute value.
The Attributes property can contain any combination of the values from the **PICTURE** enumeration.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The attribute bits were returned successfully.

E_POINTER

The address in *pdwAttr* is not valid. For example, it may be NULL.

See Also

PICTURE

IPicture::get_CurDC Quick Info

Returns the handle of the current device context. This property is valid only for bitmap pictures.

```
HRESULT get_CurDC(
  HDC* phdcOut      //Receives a pointer to device context
);
```

Parameters

phdcOut

[out] Pointer to the caller's variable to receive the device context.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The device context was returned successfully.

E_POINTER

The address in *phdcOut* is not valid. For example, it may be NULL.

Remarks

The *CurDC* property and the **IPicture::SelectPicture** method exist to circumvent restrictions in Windows; specifically, that an object can only be selected into exactly one device context at a time. In some cases, a picture object may be permanently selected into a particular device context (for example, a control may use a certain picture for a background). To use this picture property elsewhere, it must be temporarily deselected from its old device context, selected into the new device context for the operation, then reselected back into the old device context. The **IPicture::get_CurDC** method returns the device context handle into which the picture is currently selected. The **IPicture::SelectPicture** method selects the picture into a new device context, returning the old device context and the picture's GDI handle. The caller should select the picture back into the old device context when the caller is done with it, as is normal for Windows code.

Notes to Callers

The caller always owns any device contexts passed between it and the picture object. Since the picture object maintains a copy of the HDC, the caller should use a memory device context (created with the Win32 function **CreateCompatibleDC**) and not a screen device context (from **GetDC**, **CreateDC**, or **BeginPaint**), because the screen device contexts are a limited system resource.

See Also

IPicture::SelectPicture

IPicture::get_Handle Quick Info

Returns the Windows GDI handle of the picture managed within this picture object to a specified location.

```
HRESULT get_Handle(

  OLE_HANDLE* phandle      //Receives a pointer to GDI handle
);
```

Parameters

phandle

[out] Pointer to the caller's **OLE_HANDLE** variable that receives the handle. The caller is responsible for this handle upon successful return. The variable is set to NULL on failure.

Return Values

This method supports the standard return values E_FAIL and E_OUTOFMEMORY, as well as the following:

S_OK

The handle was returned successfully.

E_POINTER

The address in *phandle* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

The picture object may retain ownership of the picture; however, the caller can be assured that the picture will remain valid until either the caller specifically destroys the picture or the picture object is itself destroyed. The *fOwn* parameter to **OleCreatePictureIndirect** determines ownership when the picture object is created. **OleLoadPicture** forces *fOwn* to TRUE.

IPicture::get_Height Quick Info

Returns the current height of the picture in the picture object.

```
HRESULT get_Height(
  OLE_YSIZE_HIMETRIC* pheight      //Receives a pointer to height
);
```

Parameters

pheight

[out] Pointer to the caller's **OLE_YSIZE_HIMETRIC** variable that receives the height.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The height was returned successfully.

E_POINTER

The address in *pheight* is not valid. For example, it may be NULL.

IPicture::get_hPal Quick Info

Returns a copy of the palette currently used by the picture object.

```
HRESULT get_hPal(

  OLE_HANDLE* phpal      //Receives a pointer to palette handle
);
```

Parameters

phpal

[out] Pointer to the caller's **OLE_HANDLE** variable to receive the palette handle. The variable is set to NULL on failure.

Return Values

This method supports the standard return values E_FAIL and E_OUTOFMEMORY, as well as the following:

S_OK

The handle was returned successfully.

S_FALSE

This picture has no palette. The parameter **phpal* is set to NULL.

E_POINTER

The address in *phpal* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

If the picture object has ownership of the picture, it also has ownership of the palette and will destroy it when the object is itself destroyed. Otherwise the caller owns the palette. The *fOwn* parameter to **OleCreatePictureIndirect** determines ownership. **OleLoadPicture** sets *fOwn* to TRUE to indicate that the picture object owns the palette.

IPicture::get_KeepOriginalFormat Quick Info

Returns the current value of the picture's **KeepOriginalFormat** property.

```
HRESULT get_KeepOriginalFormat(
  BOOL* pfkeep      //Receives a pointer to the value of KeepOriginalFormat
);
```

Parameters

pfkeep

[out] Pointer to the caller's **BOOL** variable that receives the value of the property.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The value of the **KeepOriginalFormat** property was returned successfully.

E_POINTER

The address in *pfkeep* is not valid. For example, it may be NULL.

See Also

IPicture::put_KeepOriginalFormat

IPicture::get_Type Quick Info

Returns the current type of the picture contained in the picture object.

```
HRESULT get_Type(
  short* ptype      //Receives a pointer to the picture type
);
```

Parameters

ptype

[out] Pointer to the caller's **short** variable to receive the picture type. The Type property can have any one of the values contained in the **PICTYPE** enumeration.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The type was returned successfully.

E_POINTER

The address in *ptype* is not valid. For example, it may be NULL.

See Also

OleCreatePictureIndirect, **PICTYPE**

IPicture::get_Width Quick Info

Returns the current width of the picture in the picture object.

```
HRESULT get_Width(
  OLE_XSIZE_HIMETRIC* pwidth      //Receives a pointer to width
);
```

Parameters

pwidth

[out] Pointer to the caller's **OLE_XSIZE_HIMETRIC** variable that receives the width.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The width was returned successfully.

E_POINTER

The address in *pwidth* is not valid. For example, it may be NULL.

IPicture::PictureChanged Quick Info

Notifies the picture object that its picture resource has changed. On Win32, this method only sends an IPropertyNotifySink::OnChanged(DISPID_PICT_HANDLE) to any connected sinks.

HRESULT PictureChanged(void);

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

This value is returned in all cases except when the picture object is uninitialized.

IPicture::put_KeepOriginalFormat Quick Info

Sets the value of the picture's **KeepOriginalFormat** property.

```
HRESULT put_KeepOriginalFormat(
  BOOL keep      //Specifies the new value of KeepOriginalFormat
);
```

Parameters

keep

[in] Specifies the new value to assign to the property.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The property was successfully changed.

See Also

IPicture::get_KeepOriginalFormat

IPicture::Render

Quick Info

Renders (draws) a specified portion of the picture defined by the offset (*xSrc*,*ySrc*) of the source picture and the dimensions to copy (*cxSrc*,*xySrc*). This picture is rendered onto the specified device context, positioned at the point (*x*,*y*), and scaled to the dimensions (*cx*,*cy*). The *prcWBounds* parameter specifies the position of this rendering if the destination device context is itself a metafile. Such information is necessary to place one metafile in another. For more information, see the *prcWBounds* parameter of **IViewObject2::Draw**.

```
HRESULT Render(
  HDC hdc ,                          //Handle of device context on which to render the image
  long x ,                           //Horizontal position of image in hdc
  long y ,                           //Vertical position of image in hdc
  long cx ,                          //Horizontal dimension of destination rectangle
  long cy ,                          //Vertical dimension of destination rectangle
  OLE_XPOS_HIMETRIC xSrc ,           //Horizontal offset in source picture
  OLE_YPOS_HIMETRIC ySrc ,           //Vertical offset in source picture
  OLE_XSIZE_HIMETRIC cxSrc ,         //Amount to copy horizontally in source picture
  OLE_YSIZE_HIMETRIC cySrc ,         //Amount to copy vertically in source picture
  LPCRECT prcWBounds                 //Pointer to position of destination for a metafile hdc
);
```

Parameters

hdc

[in] Handle of the device context on which to render the image.

x

[in] Horizontal coordinate in *hdc* at which to place the rendered image.

y

[in] Vertical coordinate in *hdc* at which to place the rendered image.

cx

[in] Horizontal dimension of the destination rectangle.

cy

[in] Vertical dimension of the destination rectangle.

xSrc

[in] Horizontal offset in the source picture from which to start copying.

ySrc

[in] Vertical offset in the source picture from which to start copying.

cxSrc

[in] Horizontal extent to copy from the source picture.

cySrc

[in] Vertical extent to copy from the source picture.

prcWBounds

[in] Pointer to a rectangle containing the position of the destination within a metafile device context if *hdc* is a metafile DC. Cannot be NULL in such cases.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

The picture was rendered successfully.

E_POINTER

The address in *prcWBounds* is not valid when *hdc* contains a metafile device context.

IPicture::SaveAsFile Quick Info

Saves the picture's data into a stream in the same format that it would save itself into a file. Bitmaps use the BMP file format, metafiles the WMF format, and icons the ICO format. For more information, see the *Win32 Programmer's Reference*.

```
HRESULT SaveAsFile(
  IStream * pstream ,        //Pointer to stream where picture writes its data
  BOOL fSaveMemCopy ,        //Indicates whether to save the picture in memory
  LONG* pcbSize              //Receives a pointer to the number of bytes written to stream
);
```

Parameters

pstream

[in] Pointer to the stream into which the picture writes its data.

fSaveMemCopy

[in] Flag indicating whether or not to save a copy of the picture in memory.

pcbSize

[out] Pointer to the caller's **LONG** variable to receive the number of bytes written into the stream. This value can be NULL, indicating that the caller does not require this information.

Return Values

This method supports the standard return values E_FAIL and E_INVALIDARG, as well as the following:

S_OK

The picture was saved successfully.

IPicture::SelectPicture Quick Info

Selects a bitmap picture into a given device context, and returns the device context in which the picture was previously selected as well as the picture's GDI handle. This method works in conjunction with **IPicture::get_CurDC**.

```
HRESULT SelectPicture(
  HDC hdcIn ,                 //New device context
  HDC* phdcOut ,              //Receives a pointer to the previous device context
  OLE_HANDLE* phbmpOut        //Receives a pointer to GDI handle of the picture
);
```

Parameters

hdcIn

[in] Device context in which to select the picture.

phdcOut

[out] Pointer to the caller's **HDC** variable to receive the previous device context. This parameter can be NULL if the caller does not need this information. Ownership of the device context is always the responsibility of the caller.

phbmpOut

[out] Pointer to the caller's **HDC** variable to receive the GDI handle of the picture. This parameter can be NULL if the caller does not need the handle. Ownership of this handle is determined by the *fOwn* parameter passed to **OleCreatePictureIndirect**. Pictures loaded from a stream always own their resources.

Return Values

This method supports the standard return valuesE_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

The picture was selected successfully.

See Also

IPicture::get_CurDC

IPicture::set_hPal Quick Info

Assigns a GDI palette to the picture contained in the picture object.

```
HRESULT set_hPal(
  OLE_HANDLE hpal      //Handle for GDI palette for the picture
);
```

Parameters

hpal

[in] Handle to the GDI palette assigned to the picture.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

The palette was assigned successfully.

Remarks

Notes to Implementers

Ownership of the palette passed to this method depends on how the picture object was created, as specified by the *fOwn* parameter to **OleCreatePictureIndirect**. **OleLoadPicture** forces *fOwn* to TRUE; if the object owns the picture, then it takes over ownership of this palette.

IPicture - Ole Implementation

Picture objects provide a language-neutral abstraction for bitmaps, icons, and metafiles. As with the standard font object, the system provides a standard implementation of the picture object. Its primary interfaces are **IPicture** and **IPictureDisp**, the latter being derived from **IDispatch** to provide access to the picture's properties through Automation. A picture object is created with **OleCreatePictureIndirect** and supports both the **IPicture** and the **IPictureDisp** interfaces.

Remarks

The OLE-provided picture object implements the complete semantics of the **IPicture** and **IPictureDisp** interfaces.

See Also

IPicture

IPictureDisp

Quick Info

The **IPictureDisp** interface exposes the picture object's properties through Automation. It provides a subset of the functionality available through **IPicture** methods.

When to Implement

A picture object implements this interface along with **IPicture** to provide access to the picture's properties through Automation. Typically, it is not necessary to implement this interface on your own object since there is an OLE-provided picture object.

The following table describes the dispIDs for the various picture properties.

Symbol	Value
DISPID_PICT_HANDLE	0
DISPID_PICT_HPAL	2
DISPID_PICT_TYPE	3
DISPID_PICT_WIDTH	4
DISPID_PICT_HEIGHT	5
DISPID_PICT_RENDER	6

Each property in the **IPictureDisp** interface includes a get_*PropertyName* method if the property supports read access and a put_*PropertyName* method if the property supports write access. Most of the properties support read access only with the exception of the hPal property.

Property	Type	Access Allowed	Description
Handle	OLE_HANDLE (int)	R	The Windows GDI handle of the picture
hPal	OLE_HANDLE (int)	RW	The Windows handle of the palette used by the picture.
Type	short	R	The type of picture (see below).
Width	OLE_XSIZE_HIMETRIC (long)	R	The width of the picture.
Height	OLE_YSIZE_HIMETRIC (long)	R	The height of the picture.

When to Use

Use this interface to change or retrieve the properties of a picture object.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

See Also

IPicture

IPictureDisp - Ole Implementation

Picture objects provide a language-neutral abstraction for bitmaps, icons, and metafiles. As with the standard font object, the system provides a standard implementation of the picture object. Its primary interfaces are **IPicture** and **IPictureDisp**, the latter being derived from **IDispatch** to provide access to the picture's properties through Automation. A picture object is created with **OleCreatePictureIndirect** and supports both the **IPicture** and the **IPictureDisp** interfaces.

Remarks

The OLE-provided picture object implements the complete semantics of the **IPicture** and **IPictureDisp** interfaces.

See Also

IPicture, **IPictureDisp**

IPointerInactive Quick Info

The **IPointerInactive** interface enables an object to remain inactive most of the time, yet still participate in interaction with the mouse, including drag and drop.

Objects can be active (in-place or UI active) or they can be inactive (loaded or running). An active object creates a window and can receive Windows mouse and keyboard messages. An inactive object can render itself and provide a representation of its data in a given format. While they provide more functionality, active objects also consume more resources than inactive objects. Typically, they are larger and slower than inactive objects. Thus, keeping an object inactive can provide performance improvements.

However, an object, such as a control, needs to be able to control the mouse pointer, fire mouse events, and act as a drop target so it can participate in the user interface of its container application.

When to Implement

Implement this interface on an object, such as a control, so the object can support a minimal level of interaction with the mouse and keyboard while it is in the inactive state. The object can control the mouse pointer, fire mouse events, and act as a drop target without being in the active state at all times. The object does not have to set the OLEMISC_ACTIVATEWHENVISIBLE enumeration value, does not have to have a window, and thus, can increase its performance.

If the object must work with down-level containers, it may have to set the OLEMISC_ACTIVATEWHENVISIBLE enumeration value. However, an updated container that supports objects that implement **IPointerInactive** can use the OLEMISC_IGNOREACTIVATEWHENVISIBLE enumeration value to override OLEMISC_ACTIVATEWHENVISIBLE.

When to Use

A container calls the methods in this interface for its embedded objects so that the embedded objects can participate in the user interface for the application.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns a pointer to a specified interface.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IPointerInactive Methods	Description
GetActivationPolicy	Returns the present activation policy for the object.
OnInactiveMouseMove	Notifies the object that the mouse pointer has moved over it so the object can fire mouse events.
OnInactiveSetCursor	Sets the mouse pointer for an inactive object.

See Also

OLEMISC

IPointerInactive::GetActivationPolicy Quick Info

Returns the present activation policy for the object. This method is called by the container on receipt of a WM_SETCURSOR or WM_MOUSEMOVE message when an inactive object is under the mouse pointer.

```
HRESULT GetActivationPolicy(

  DWORD* pdwPolicy              //Pointer to activation policy
);
```

Parameter

pdwPolicy

[out] Pointer to the activation policy as specified by the **POINTERINACTIVE** enumeration values.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The activation policy was successfully returned.

Remarks

A container calls this method when it receives a WM_SETCURSOR or WM_MOUSEMOVE message with the mouse pointer over an inactive object that supports **IPointerInactive**. The object returns its activation policy through the **POINTERINACTIVE** enumeration values.

The object can request to be in-place activated as soon as the mouse enters it through the POINTERINACTIVE_ACTIVATEONENTRY value. An object that provides more visual feedback than simply setting the mouse pointer would use this value. For example, if the object supports special visual feedback, it must enter the active state so it can draw the visual feedback that it supports.

An object can also use this method to request activation when the mouse is dragged over them during a drag and drop operation through the POINTERINACTIVE_ACTIVATEONDRAG. See the **POINTERINACTIVE** enumeration for more information.

If the object returns one of these values, the container should activate the object immediately and forward the Window message that triggered the call. The object then stays active and processes subsequent messages through its own window until the container gets another WM_SETCURSOR or WM_MOUSEMOVE. At this point, the container should deactivate the object.

Note For windowless OLE objects this mechanism is slightly different. See **IOleInPlaceSiteWindowless** for more information on drag and drop operations for windowless objects.

If the object returns both the POINTERINACTIVE_ACTIVATEONENTRY and the POINTERINACTIVE_DEACTIVATEONLEAVE values, the object is activated only when the mouse is over the object. If the POINTERINACTIVE_ACTIVATEONENTRY value alone is set, the object is activated once when the mouse first enters it, and it remains active.

Note to Callers

The activation policy should not be cached. The container should call this method each time the mouse enters an inactive object.

See Also

IOleInPlaceSiteWindowless, **POINTERINACTIVE**

IPointerInactive::OnInactiveMouseMove Quick Info

Notifies the object that the mouse pointer has moved over it so the object can fire mouse events. This method is called by the container on receipt of a WM_MOUSEMOVE method when an inactive object is under the mouse pointer.

```
HRESULT OnInactiveMouseMove(
    LPCRECT pRectBounds,      //Object bounding rectangle
    LONG x,                   //Horizontal coordinate
    LONG y,                   //Vertical coordinate
    DWORD grfKeyState         //
);
```

Parameter

pRectBounds

[in] The object bounding rectangle, in client coordinates of the containing window. This parameter tells the object its exact position and size on the screen when the WM_MOUSEMOVE message was received. This value is specified in units of the client's coordinate system.

x

[in] Horizontal coordinate of mouse location in units of the client's containing window.

y

[in] Vertical coordinate of mouse location in units of the client's containing window.

grfKeyState

[in] Identifies the current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the values MK_CONTROL, MK_SHIFT, MK_ALT, MK_BUTTON, MK_LBUTTON, MK_MBUTTON, and MK_RBUTTON.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The mouse pointer was successfully set.

Remarks

The container calls this method to notify the object that the mouse pointer is over the object after checking the object's activation policy by calling the **IPointerInactive::GetActivationPolicy** method. If the object has not requested to be activated in-place through that call, the container dispatches subsequent WM_MOUSEMOVE messages to the inactive object by calling **OnInactiveMouseMove** as long as the mouse pointer stays over the object. The object can then fire mouse move events.

To avoid rounding errors and to make the job easier on the object implementer, this method takes window coordinates in the units of its containing client window, that is, the window in which the object is displayed, instead of the usual HIMETRIC units. Thus, the same coordinates and code path can be used when the object is active and inactive. The window coordinates specify the mouse position. The bounding rectangle is also specified in the same coordinate system.

See Also

IPointerInactive::GetActivationPolicy

IPointerInactive::OnInactiveSetCursor Quick Info

Sets the mouse pointer for an inactive object. This method is called by the container on receipt of a WM_SETCURSOR method when an inactive object is under the mouse pointer.

```
HRESULT OnInactiveSetCursor(
    LPCRECT pRectBounds,      //Object bounding rectangle
    LONG x,                   //Horizontal coordinate
    LONG y,                   //Vertical coordinate
    DWORD dwMouseMsg,         //Mouse message identifier
    BOOL fSetAlways           //Indicates whether object must set the mouse pointer
);
```

Parameter

pRectBounds

[in] The object bounding rectangle specified in client coordinate units of the containing window. This parameter tells the object its exact position and size on the screen when the WM_SETCURSOR message was received. This value is specified in units of the client's coordinate system.

x

[in] Horizontal coordinate of mouse location in units of the client's containing window.

y

[in] Vertical coordinate of mouse location in units of the client's containing window.

dwMouseMsg

[in] Identifier of the mouse message for which a WM_SETCURSOR occurred.

fSetAlways

[in] If this value is TRUE, the object must set the cursor; if this value is FALSE, the object is not obligated to set the cursor, and should return S_FALSE in that case.

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

The mouse pointer was successfully set.

S_FALSE

The object did not set the cursor; the container should either set the cursor or call the object again with the parameter *fSetAlways* set to TRUE.

Remarks

The container calls this method to set the mouse pointer over an inactive object after checking the object's activation policy by calling the **IPointerInactive::GetActivationPolicy** method. If the object has not requested to be activated in-place through that call, the container dispatches subsequent WM_SETCURSOR messages to the inactive object by calling **OnInactiveSetCursor** as long as the

mouse pointer stays over the object.

To avoid rounding errors and to make the job easier on the object implementer, this method takes window coordinates in the units of its containing client window, that is, the window in which the object is displayed, instead of the usual HIMETRIC units. Thus, the same coordinates and code path can be used when the object is active and inactive. The window coordinates specify the mouse position. The bounding rectangle is also specified in the same coordinate system.

OnInactiveSetCursor takes an additional parameter (*fSetAlways*) indicating whether the object is obligated to set the cursor or not. Containers should first call this method with this parameter FALSE. The object may return S_FALSE to indicate that it did not set the cursor. In that case, the container should either set the cursor itself, or, if it does not wish to do this, call the **OnInactiveSetCursor** method again with *fSetAlways* being TRUE.

See Also

IPointerInactive::GetActivationPolicy

IProgressNotify

The **IProgressNotify** interface enables applications and other objects to receive notifications of changes in the progress of a downloading operation.

When to Implement

You do not need to implement **IProgressNotify**. The downloading code implements this interface, and the asynchronous storage implementation provides a connection point for dispatching notifications to it. An application can also register an **IProgressNotify** sink to receive progress notifications for individual streams.

When to Use

You do not need to call this interface. The Compound Files implementation uses **IProgressNotify::OnProgress** to control the blocking behavior of the asynchronous storage and to trigger additional byte range requests if applicable.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IProgressNotify Method	Description
OnProgress	Receives status information about progress of download operation.

IProgressNotify::OnProgress

Notifies registered objects and applications of the progress of a downloading operation.

HRESULT OnProgress(

DWORD *dwProgressCurrent* // Amount of data available
DWORD *dwProgressMaximum* // Total amount of data to be downloaded
BOOL *fAccurate* // Reliability of notifications
BOOL *fOwner* // Ownership of blocking behavior
);

Parameters

dwProgressCurrent

[in] The amount of data available.

dwProgressMaximum

[in] The total amount of data to be downloaded.

fAccurate

[in] Values in *dwProgressCurrent* and *dwProgressMaximum* are either reliable (TRUE) or unreliable (FALSE). The FALSE value indicates that control structures for determining the actual position of, or amount of, data yet to be downloaded are not available.

fOwner

[in] Indicates whether this **OnProgress** call can control the blocking behavior of the operation. If TRUE, the caller can use return values from **OnProgress** to block (STG_S_BLOCK), retry (STG_S_RETRYNOW), or monitor (STG_S_MONITORING) the operation. If FALSE, the return value from **OnProgress** has no influence on blocking behavior.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as the following:

STG_S_RETRYNOW

The caller is to retry the operation immediately. (This value is most useful for applications that do blocking from within the callback routine.)

STG_S_BLOCK

The caller is to block the download and retry the call as needed to determine if additional data is available. This is the default behavior if no sinks are registered on the connection point.

STG_S_MONITORING

The callback recipient reliquishes control of the downloading process to one of the other objects or applications that have registered progress notification sinks on the same stream. This is useful if the notification sink is interested only in gathering statistics.

E_PENDING

Data is currently unavailable. The caller is to try again after some desired interval. The notification sink returns this value if the asynchronous storage is to operate in nonblocking mode.

Remarks

Sinks may be inherited by any substorage or substream of a given storage. If no sink is registered, the thread will block until the requested data becomes available, or the download is canceled by the downloader.

Where multiple objects or applications have registered progress notification sinks on a single stream, only one of them can control the behavior of a download. Ownership of the download goes to:

1. The first sink to register with the storage or stream.
2. Any advise skinks that may have been inherited from the parent storage (if the storage was created with ASYNC_MODE_COMPATIBILITY.

Any one of the sinks can relinquish control to the next connection point by returning STG_S_MONITORING to the connection point making the current caller. Once a connection point obtains control (through receiving STG_S_BLOCK or STG_S_RETRYNOW), all subsequent connection points calling **IProgressNotify::OnProgress** will set *fOwner* to FALSE.

IPropertyBag

The **IPropertyBag** interface provides an object with a property bag in which the object can persistently save its properties.

When a client wishes to have exact control over how individually named properties of an object are saved, it would attempt to use an object's **IPersistPropertyBag** interface as a persistence mechanism. In that case the client supplies a property bag to the object in the form of an **IPropertyBag** interface.

When the object wishes to read a property in **IPersistPropertyBag::Load** it will call **IPropertyBag::Read**. When the object is saving properties in **IPersistPropertyBag::Save** it will call **IPropertyBag::Write**. Each property is described with a name in *pszPropName* whose value is exchanged in a **VARIANT**. This information allows a client to save the property values as text, for instance, which is the primary reason why a client might choose to support **IPersistPropertyBag**.

The client records errors that occur during reading into the supplied error log.

When to Implement

A container implements this interface to provide its object with a way to store their properties persistently.

When to Use

An object calls the methods on this interface to read and write its properties into the container provided property bag.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPropertyBag Methods	Description
Read	Called by the control to read a property from the storage provided by the container.
Write	Called by the control to write each property in turn to the storage provided by the container.

See Also

IErrorLog, IPersistPropertyBag

IPropertyBag::Read

Called by the control to read a property from the storage provided by the container.

```
HRESULT Read(
  LPCOLESTR pszPropName,      //Pointer to the property to be read
  VARIANT* pVar,              //Pointer to the VARIANT to receive the property value
  IErrorLog* pErrorLog        //Pointer to the caller's error log
);
```

Parameters

pszPropName

[in] Pointer to the name of the property to read. Cannot be NULL.

pVar

[in, out] Pointer to the caller-initialized **VARIANT** that is to receive the property value on output. The method must set both type and value fields in the **VARIANT** before returning. If the caller initialized the **pVar->vt** field on entry, the property bag should attempt to coerce the value it knows into this type. If the caller sets **pVar->vt** to VT_EMPTY, the property bag can use whatever type is convenient.

pErrorLog

[in] Pointer to the caller's **IErrorLog** interface in which the property bag stores any errors that occur during reads. Can be NULL in which case the caller is not interested in errors.

Return Values

S_OK

The property was read successfully. The caller becomes responsible for any allocations that are contained in the **VARIANT** in *pVar*.

E_POINTER

The address in *pszPropName* is not valid (such as NULL).

E_INVALIDARG

The property named with *pszPropName* does not exist in the property bag.

E_FAIL

The property bag was unable to read the specified property, such as if the caller specified a data type to which the property bag could not coerce the known value. If the caller supplied an error log, a more descriptive error was sent there.

Remarks

This method asks the property bag to read the property named with *pszPropName* into the caller-initialized **VARIANT** in *pVar*. Errors that occur are logged in the error log pointed to by *pErrorLog.* When **pVar->vt** specifies another object pointer (VT_UNKNOWN) then the property bag is responsible for creating and initializing the object described by *pszPropName*.

E_NOTIMPL is not a valid return code since any object implementing this interface must support the entire functionality of the interface.

See Also

IPropertyBag::Write

IPropertyBag::Write

Called by the control to write each property in turn to the storage provided by the container.

```
HRESULT Write(
  LPCOLESTR pszPropName,      //Points to the property to be written
  VARIANT* pVar               //Points to the VARIANT containing the property value and type
);
```

Parameters

pszPropName

[in] Pointer to the name of the property to write. Cannot be NULL.

pVar

[in] Pointer to the caller-initialized **VARIANT** that holds the property value to save. The caller owns this **VARIANT** and is responsible for all allocations therein. That is, the property bag itself does not attempt to free data in the **VARIANT**.

Return Values

S_OK

The property bag successfully saved the requested property.

E_FAIL

There was a problem writing the property. It is possible that the property bag does not understand how to save a particular **VARIANT** type.

E_POINTER

The address in *pszPropName* or *pVar* is not valid (such as NULL). The caller must supply both.

Remarks

This method asks the property bag to save the property named with *pszPropName* using the type and value in the caller-initialized **VARIANT** in *pVar.* In some cases the caller may be asking the property bag to save another object, that is, when **pVar->vt** is VT_UNKNOWN. In such cases, the property bag queries this object pointer for some persistence interface, like **IPersistStream** or even **IPersistPropertyBag** again and has that object save its data as well. Usually, this results in the property bag having some byte array for this object which can be saved as encoded text (hex string, MIME, etc.). When the property bag is later used to reinitialize a control, the client that owns the property bag must recreate the object when the caller asks for it, initializing that object with the previously saved bits.

This allows very efficient persistence operations for large BLOB properties like a picture, where the owner of the property bag itself directly asks the picture object (managed as a property in the control being saved) to save into a specific location. This avoids potential extra copy operations that would be involved with other property-based persistence mechanisms.

E_NOTIMPL is not a valid return code as any object implementing this interface must support the entire functionality of the interface.

See Also

IPropertyBag::Read

IPropertyNotifySink Quick Info

The **IPropertyNotifySink** interface is implemented by a sink object to receive notifications about property changes from an object that supports **IPropertyNotifySink** as an "outgoing" interface. The client that needs to receive the notifications in this interface (from a supporting connectable object) creates a sink with this interface and connects it to the connectable object through the connection point mechanism. For more information on connection points, see **IConnectionPointContainer**.

The object is itself required to call the methods of **IPropertyNotifySink** only for those properties marked with the **[bindable]** and **[requestedit]** attributes in the object's type information. When the object changes a **[bindable]** property, it is required to call **IPropertyNotifySink::OnChanged**. When the object is about to change a **[requestedit]** property, it must call **IPropertyNotifySink::OnRequestEdit** before changing the property and must also honor the action specified by the sink on return from this call.

The one exception to this rule is that no notifications are sent as a result of an object's initialization or loading procedures. At initialization time, it is assumed that all properties change and that all must be allowed to change. Notifications to this interface are therefore meaningful only in the context of a fully initialized/loaded object.

When to Implement

Implement this interface for a sink object that receives notifications about property changes from an object that supports this kind of notification.

When to Use

Use this interface to notify a sink object about changes in a property.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPropertyNotifySink Methods	Description
OnChanged	Notifies a sink that a bindable property has changed.
OnRequestEdit	Notifies a sink that a requestedit property is about to change.

See Also

IConnectionPoint, IConnectionPointContainer

Also, see the Automation branch of the help file for further explanations of type information, ODL files, and type libraries.

IPropertyNotifySink::OnChanged Quick Info

Notifies a sink that the **[bindable]** property specified by *dispID* has changed. If *dispID* is DISPID_UNKNOWN, then multiple properties have changed together. The client (owner of the sink) should then retrieve the current value of each property of interest from the object that generated the notification.

```
HRESULT OnChanged(
  DISPID dispID      //Dispatch identifier of the property that changed
);
```

Parameters

dispID

[in] Dispatch identifier of the property that changed, or DISPID_UNKNOWN if multiple properties have changed.

Return Values

S_OK

This return value is returned in all cases.

Remarks

S_OK is returned in all cases even when the sink does not need [bindable] properties or when some other failure has occurred. In short, the calling object simply sends the notification and cannot attempt to use an error code (such as E_NOTIMPL) to determine whether to not send the notification in the future. Such semantics are not part of this interface.

See Also

IPropertyNotifySink::OnRequestEdit

IPropertyNotifySink::OnRequestEdit Quick Info

Notifies a sink that a **[requestedit]** property is about to change and that the object is asking the sink how to proceed.

```
HRESULT OnRequestEdit(
  DISPID dispID      //Dispatch identifier of the property that is about to change
);
```

Parameters

dispID

[in] Dispatch identifier of the property that is about to change or DISPID_UNKNOWN if multiple properties are about to change.

Return Values

S_OK

The specified property or properties are allowed to change.

S_FALSE

The specified property or properties are not allowed to change. The caller must obey this return value by discarding the new property value(s). This is part of the contract of the [requestedit] attribute and this method.

Remarks

The sink may choose to allow or disallow the change to take place. For example, the sink may enforce a read-only state on the property. DISPID_UNKNOWN is a valid parameter to this method to indicate that multiple properties are about to change. In this case, the sink can enforce a global read-only state for all **[requestedit]** properties in the object, including any specific ones that the sink otherwise recognizes.

If the sink allows changes, the object must also make **IPropertyNotifySink::OnChanged** notifications for any properties that are marked **[bindable]** in addition to **[requestedit]**.

This method cannot be used to implement any sort of data validation. At the time of the call, the desired new value of the property is unavailable and thus cannot be validated. This method's only purpose is to allow the sink to enforce a read-only state on a property.

See Also

IPropertyNotifySink::OnChanged

IPropertyPage Quick Info

The **IPropertyPage** interface provides the main features of a property page object that manages a particular page within a property sheet. A property page implements at least **IPropertyPage** and can optionally implement **IPropertyPage2** if selection of a specific property is supported. See **IPerPropertyBrowsing::MapPropertyToPage** for more information on specific property browsing. The methods of **IPropertyPage2** allow the property sheet or property frame to instruct the page when to perform specific actions, mostly based on user input such as switching between pages or pressing various buttons that the frame itself manages in the dialog box.

A property page manages a dialog box that contains only those controls that should be displayed for that one page within the property sheet itself. This means that the dialog box template used to define the page should only carry the WS_CHILD style and no others. It should not include any style related to a frame, caption, or system menus or controls.

When to Implement

Implement this interface on a property page object.

When to Use

Use this interface to manage a property page object. Typically, this method is called within the OLE-supplied property frame created through **OleCreatePropertyFrame** or **OleCreatePropertyFrameIndirect**. Using the methods in this interface, the frame can display the properties and process end-user changes to the property values.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPropertyPage Methods	Description
SetPageSite	Initializes a property page and provides the page with a pointer to the **IPropertyPageSite** interface through which the property page communicates with the property frame.
Activate	Creates the dialog box window for the property page.
Deactivate	Destroys the window created with **Activate**.
GetPageInfo	Returns information about the property page.
SetObjects	Provides the property page with an array of **IUnknown** pointers for objects associated with this property page.
Show	Makes the property page dialog box visible or invisible.
Move	Positions and resizes the property

	page dialog box within the frame.
IsPageDirty	Indicates whether the property page has changed since activated or since the most recent call to **Apply**.
Apply	Applies current property page values to underlying objects specified through **SetObjects**.
Help	Invokes help in response to end-user request.
TranslateAccelerator	Provides a pointer to a **MSG** structure that specifies a keystroke to process.

See Also

IPerPropertyBrowsing, **IPropertyPage2**, **IPropertyPageSite**, **ISpecifyPropertyPages**, **OleCreatePropertyFrame**, **OleCreatePropertyFrameIndirect**

IPropertyPage::Activate Quick Info

Creates the dialog box for the property page (without a frame, caption, or system menu/controls) using *hWndParent* as the parent window and *prc* as the positioning rectangle. The *bModal* flag indicates the modality of the dialog box frame (in the current implementation of **OleCreatePropertyFrame** and **OleCreatePropertyFrameIndirect**, this parameter is always TRUE). The text in the dialog should match the locale obtained through **IPropertyPageSite::GetLocaleID**.

The property page maintains the window handle created in this process, which it uses to destroy the dialog box within **IPropertyPage::Deactivate**.

```
HRESULT Activate(
    HWND hWndParent ,      //Parent window handle
    LPCRECT prc ,          //Pointer to RECT structure
    BOOL bModal            //Dialog box frame is modal or modeless
);
```

Parameters

hWndParent

[in] Window handle of the parent of the dialog box that is being created.

prc

[in] Pointer to the **RECT** structure containing the positioning information for the dialog box. This method must create its dialog box with the placement and dimensions described by this rectangle, that is, origin point at (*prc->left*, *prc->top*) and dimensions of (*prc->right-prc->Left*, *prc->bottom-prc->top)*.

bModal

[in] Indicates whether the dialog box frame is modal (TRUE) or modeless (FALSE).

Return Values

This method supports the standard return values E_OUTOFMEMORY and

E_UNEXPECTED, as well as the following:

S_OK

The page dialog box was created successfully.

E_POINTER

The address in *prc* is not valid. For example, it may be NULL.

Remarks

Notes to Implementers

E_NOTIMPL is not a valid return value.

See Also

IPropertyPage::Activate

IPropertyPage::Apply Quick Info

Applies the current values to the underlying objects associated with the property page as previously passed to **IPropertyPage::SetObjects**.

HRESULT Apply(void);

Return Values

This method supports the standard return values E_OUTOFMEMORY and

E_UNEXPECTED, as well as the following:

S_OK

Changes were successfully applied and the property page is current with the underlying objects.

S_FALSE

Changes were applied, but the property page cannot determine if its state is current with the objects.

Remarks

The objects to be changed are provided through a previous call to **IPropertyPage::SetObjects**. By calling **IPropertyPage::SetObjects** prior to calling this method, the caller ensures that all underlying objects have the correct interfaces through which to communicate changes. Therefore, this method should not fail because of non-existent interfaces.

After applying its values, the property page should determine if its state is now current with the objects in order to properly implement **IPropertyPage::IsPageDirty** and to provide both S_OK and S_FALSE return values.

Notes to Implementers

E_NOTIMPL is not a valid return value.

See Also

IPropertyPage::IsPageDirty, **IPropertyPage::SetObjects**

IPropertyPage::Deactivate Quick Info

Destroys the window created in **IPropertyPage::Activate**.

HRESULT Deactivate(void);

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The dialog was successfully destroyed.

Remarks

Notes to Implementers

It is important that property pages not keep the dialog box around as an optimization. In a property sheet with many property pages, memory consumption would become excessive if all property pages kept their dialog boxes created at all times. Destroying the dialog box prevents excessive memory consumption due to a very large number of created controls in the dialog boxes. If the frame wishes to keep pages alive while they are not visible, it can use **IPropertyPage::Show** for that purpose. The decision is ultimately left to the frame.

E_NOTIMPL is not a valid return value.

See Also

IPropertyPage::Activate

IPropertyPage::GetPageInfo Quick Info

Fills a caller-allocated **PROPPAGEINFO** structure to provide the caller with information about the property page.

```
HRESULT GetPageInfo(
  PROPPAGEINFO *pPageInfo      //Receives a pointer to property page information structure
);
```

Parameters

pPageInfo

[out] Pointer to the caller-allocated **PROPPAGEINFO** structure in which the property page stores its page information. All allocations stored in this structure become the responsibility of the caller.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The structure was successfully filled.

E_POINTER

The address in *pPageInfo* is not valid. For example, it may be NULL.

Remarks

Notes to Implementers

E_NOTIMPL is not a valid return value.

See Also

PROPPAGEINFO

IPropertyPage::Help Quick Info

Invokes the property page help in response to an end-user request.

```
HRESULT Help(
  LPCOLESTR pszHelpDir      //Pointer to string from HelpDir key
);
```

Parameters

pszHelpDir

[in] Pointer to the string under the **HelpDir** key in the property page's CLSID information in the registry. If **HelpDir** does not exist, this will be the path found in the **InProcServer32** entry minus the server file name. (Note that **LocalServer32** is not checked in the current implementation, since local property pages are not currently supported).

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The page displayed its own help.

E_NOTIMPL

Help is either not provided or is provided only through the information in **PROPPAGEINFO**.

Remarks

Notes to Callers

Calls to this method must occur between calls to **IPropertyPage::Activate** and **IPropertyPage::Deactivate**.

Notes to Implementers

If the page fails this method (such as E_NOTIMPL), then the frame will attempt to use the *pszHelpFile* and *dwHelpContext* fields of the **PROPPAGEINFO** structure obtained through **IPropertyPage::GetPageInfo**. Therefore, the page should either implement **IPropertyPage::Help** or return help information through **IPropertyPage::GetPageInfo**.

See Also

IPropertyPage::Activate, **IPropertyPage::Deactivate**, **IPropertyPage::GetPageInfo**, **PROPPAGEINFO**

IPropertyPage::IsPageDirty Quick Info

Indicates whether the property page has changed its state since activation or since the last call to **IPropertyPage::Apply**. The property sheet uses this information to enable or disable the Apply button in the dialog box. There is no need to apply the values on a property page if those values are already current with the underlying objects.

HRESULT IsPageDirty(void);

Return Values

S_OK

The value state of the property page is dirty, that is, it has changed and is different from the state of the objects.

S_FALSE

The value state of the page has not changed and is current with that of the objects.

Remarks

Notes to Implementers

This method has no reason to return an error code, since the inability to determine if the page is dirty should return S_OK as a default. In this way, the user has a chance to update the values. The page should not return an error code, since an error code is not the same as S_OK and would indicate that the page is not dirty. Then, the property frame could potentially disable the Apply button, not allowing the user to make sure that the property values are current.

See Also

IPropertyPage::Apply

IPropertyPage::Move Quick Info

Repositions and resizes the property page dialog box according to the contents of *prc*. The rectangle specified by *prc* is treated identically to that passed to **IPropertyPage::Activate**.

```
HRESULT Move(
  LPCRECT prc      //Pointer to RECT structure
);
```

Parameters

prc

[in] Pointer to the **RECT** structure containing the positioning information for the page dialog box.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The page repositioned itself successfully.

E_POINTER

The address in *prc* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

Calls to this method must occur after a call to **IPropertyPage::Activate** and before a corresponding call to **IPropertyPage::Deactivate**.

Notes to Implementers

The page must create its dialog box with the placement and dimensions described by this rectangle, that is, origin point at (*prc->left*, *prc->top*) and dimensions of (*prc->right-prc->Left*, *prc->bottom-prc->top*).

See Also

IPropertyPage::Activate, IPropertyPage::Deactivate

IPropertyPage::SetObjects Quick Info

Provides the **IUnknown** pointers of the objects affected by the property sheet in which this property page is displayed. When the property page receives a call to **IPropertyPage::Apply**, it must send value changes to these objects through whatever interfaces are appropriate. The property page must query for those interfaces. This method can fail if the objects do not support the interfaces expected by the property page.

```
HRESULT SetObjects(
  ULONG cObjects ,      //Number of IUnknown pointers in the ppUnk array
  IUnknown **ppUnk      //Pointer to array
);
```

Parameters

cObjects

[in] Number of **IUnknown** pointers in the array pointed to by *ppUnk*. If zero, the property page must release any pointers previously passed to this method.

ppUnk

[in] Pointer to an array of **IUnknown** interface pointers where each pointer identifies a unique object affected by the property sheet in which this (and possibly other) property pages are displayed. The property page must cache these pointers calling **IUnknown::AddRef** for each pointer at that time. This array of pointers is the same one that was passed to **OleCreatePropertyFrame** or **OleCreatePropertyFrameIndirect** to invoke the property sheet.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG,

E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The property page successfully saved the pointers it needed.

E_NOINTERFACE

One of the objects in *ppUnk* did not support the interface expected by this property page, and so this property page cannot communicate with it.

E_POINTER

The address in *ppUnk* is not valid. For example, it may be NULL.

Remarks

The property page is required to keep the pointers returned by this method or others queried through them. If these specific **IUnknown** pointers are held, the property page must call **IUnknown::AddRef** through each when caching them, until the time when **IPropertyPage::SetObjects** is called with *cObjects* equal to zero. At that time, the property page must call **IUnknown::Release** through each pointer, releasing any objects that it held.

The caller must provide the property page with these objects before calling **IPropertyPage::Activate**, and

should call **IPropertyPage::SetObjects** with zero as the parameter when deactivating the page or when releasing the object entirely. Each call to **SetObjects** with a non-NULL *ppUnk* parameter must be matched with a later call to **SetObjects** with zero in the *cObjects* parameter.

Notes to Implementers

E_NOTIMPL is not a valid return value.

See Also

IPropertyPage::Activate, **OCPFIPARAMS**, **OleCreatePropertyFrame**, **OleCreatePropertyFrameIndirect**

IPropertyPage::SetPageSite Quick Info

Initializes a property page and provides the property page object with the **IPropertyPageSite** interface through which the property page communicates with the property frame.

```
HRESULT SetPageSite(

  IPropertyPageSite *pPageSite      //Pointer to the site object
);
```

Parameters

pPageSite

[in] Pointer to the **IPropertyPageSite** interface of the page site that manages and provides services to this property page within the entire property sheet.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The page site was saved and the page object was fully initialized.

Remarks

Notes to Implementers

If the *pPageSite* parameter is NULL, this method must call `pPageSite->Release` on any **IPropertyPageSite** pointer passed during a previous call to this method. If non-NULL, this method must save the **IPropertyPageSite** pointer value and call `pPageSite->AddRef`. Two consecutive calls to this method with a non-NULL site pointer are not allowed and should cause the property page to return E_UNEXPECTED.

E_NOTIMPL is not a valid return value. All property pages must implement this method.

See Also

IPropertyPageSite

IPropertyPage::Show Quick Info

Makes the property page dialog box visible or invisible according to the *nCmdShow* parameter. If the page is made visible, the page should set the focus to itself, specifically to the first property on the page.

```
HRESULT Show(
  UINT nCmdShow      //Indicates whether to make the page visible or hidden
);
```

Parameters

nCmdShow

[in] Command describing whether to become visible (SW_SHOW or SW_SHOWNORMAL) or hidden (SW_HIDE). No other values are valid for this parameter.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The show command was successfully invoked.

Remarks

Notes to Callers

Calls to this method must occur after a call to **IPropertyPage::Activate** and before a corresponding call to **IPropertyPage::Deactivate**.

Notes to Implementers

E_NOTIMPL is not a valid return value. E_OUTOFMEMORY is not a valid return value, since no memory should be used in implementing this method.

See Also

IPropertyPage::Activate, **IPropertyPage::Deactivate**

IPropertyPage::TranslateAccelerator Quick Info

Instructs the property page to process the keystroke described in *pMsg*.

```
HRESULT TranslateAccelerator(
  LPMSG pMsg      //Pointer to MSG structure
);
```

Parameters

pMsg

[in] Pointer to the **MSG** structure describing the keystroke to process.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The property page handles the accelerator.

S_FALSE

The property page handles accelerators, but this one was not useful to it.

E_NOTIMPL

The property page does not handle accelerators.

E_POINTER

The address in *pMsg* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

Calls to this method must occur after a call to **IPropertyPage::Activate** and before the corresponding call to **IPropertyPage::Deactivate**.

See Also

IPropertyPage::Activate, IPropertyPage::Deactivate

IPropertyPage2 Quick Info

The **IPropertyPage2** interface is an extension to **IPropertyPage** to support initial selection of a property on a page. It works in conjunction with the implementation of **IPerPropertyBrowsing::MapPropertyToPage** on an object that supplies properties and specifies property pages through **ISpecifyPropertyPages**. This interface has only one extra method in addition to those in **IPropertyPage**. That method, **IPropertyPage2::EditProperty** tells the page which property to highlight.

When to Implement

Implement this interface if your property page object supports selection of a specific property.

When to Use

Use this interface to select a specific property in a property page.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPropertyPage Methods	Description
SetPageSite	Initializes a property page and provides the page with a pointer to the **IPropertyPageSite** interface through which the property page communicates with the property frame.
Activate	Creates the dialog box window for the property page.
Deactivate	Destroys the window created with **Activate**.
GetPageInfo	Returns information about the property page.
SetObjects	Provides the property page with an array of **IUnknown** pointers for objects associated with this property page.
Show	Makes the property page dialog box visible or invisible.
Move	Positions and resizes the property page dialog box within the frame.
IsPageDirty	Indicates whether the property page has changed since activated or since the most recent call to **Apply**.
Apply	Applies current property page values to underlying objects specified through **SetObjects**.
Help	Invokes help in response to end-user

	request.
TranslateAccelerator	Provides a pointer to a **MSG** structure that specifies a keystroke to process.

IPropertyPage2 Methods	Description
EditProperty	Specifies which field is to receive the focus when the property page is activated.

See Also

IPerPropertyBrowsing, IPropertyPage, IPropertyPageSite, ISpecifyPropertyPages, IPerPropertyBrowsing::MapPropertyToPage

IPropertyPage2::EditProperty Quick Info

Specifies in *dispID* the property's control on the property page to receive the focus when the page is activated.

```
HRESULT EditProperty(
  DISPID dispID      //Dispatch identifier for property
);
```

Parameters

dispID

[in] Identifies the property that is to receive the focus.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The property was successfully highlighted.

E_NOTIMPL

This method is not currently implemented; the interface is probably provided in anticipation of future work on this page.

Remarks

Notes to Implementers

If this method is called before a page is activated, the page should store the property and set the focus to it in the next call to **IPropertyPage::Activate**. If the page is already active, **IPropertyPage2::EditProperty** should set the focus to the specific property field.

See Also

IPropertyPage::Activate

IPropertyPageSite Quick Info

The **IPropertyPageSite** interface provides the main features for a property page site object. For each property page created within a property frame, the frame creates a property page site to provide information to the property page and to receive notifications from the page when changes occur. This latter notification is used to initiate a call to **IPropertyPage::IsPageDirty**, the return value of which is then used to enable or disable the frame's Apply button.

When to Implement

Implement this interface on a site object that will manage a property page on behalf of the property frame. Typically, the OLE-provided property frame created through **OleCreatePropertyFrame** and **OleCreatePropertyFrameIndirect** implements site objects.

When to Use

Use a site object with this interface to set up communications between the property frame and the property page object.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPropertyPageSite Methods	Description
OnStatusChange	Indicates that the user has modified property values on the property page.
GetLocaleID	Returns the locale identifier so the property page can adjust itself to country-specific settings.
GetPageContainer	Returns an **IUnknown** pointer for the object representing the entire property frame dialog box that contains all the pages.
TranslateAccelerator	Passes a keystroke to the property frame for processing.

See Also

IPerPropertyBrowsing, IPropertyPage, IPropertyPage2, IPropertyPageSite - Ole Implementation, ISpecifyPropertyPages

IPropertyPageSite::GetLocaleID Quick Info

Returns the locale identifier (an LCID) that a property page can use to adjust itself to the language in use and other country-specific settings.

```
HRESULT GetLocaleID(
    LCID* pLocaleID      //Receives a pointer to the locale identifier
);
```

Parameters

pLocaleID

[out] Pointer to locale identifier.

Return Values

S_OK

The locale was returned successfully.

E_POINTER

The address in *pLocaleID* is not valid. For example, it may be NULL.

See Also

OCPFIPARAMS, **PROPPAGEINFO**

IPropertyPageSite::GetPageContainer Quick Info

Returns an **IUnknown** pointer to the object representing the entire property frame dialog box that contains all the pages. Calling this method could potentially allow one page to navigate to another.

However, there are no "container" interfaces currently defined for this role, so this method always fails in the current property frame implementation.

```
HRESULT GetPageContainer(
  IUnknown** ppUnk      //Indirect pointer to the interface of the container object
);
```

Parameters

ppUnk

[out] Indirect pointer to the **IUnknown** interface on the container objectSet to NULL on failure.

Return Values

E_NOTIMPL

This is the only return value allowed at this time.

IPropertyPageSite::OnStatusChange Quick Info

Informs the frame that the property page managed by this site has changed its state, that is, one or more property values have been changed in the page. Property pages should call this method whenever changes occur in their dialog boxes.

```
HRESULT OnStatusChange(
    DWORD dwFlags      //Indicates what changes have occurred
);
```

Parameters

dwFlags

[in] Flags to indicate what changes have occurred. The *dwFlags* parameter can contain either of these two values to indicate the type of status change:

Flag Value	Description
PROPPAGESTATUS_DIRTY	Values in pages have changed so the state of the Apply button should be updated.
PROPPAGESTATUS_VALIDATE	Now is an appropriate time to apply changes.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The status change was noted.

IPropertyPageSite::TranslateAccelerator Quick Info

Instructs the page site to process a keystroke if it desires.

```
HRESULT TranslateAccelerator(
  LPMSG pMsg      //Pointer to MSG structure
);
```

Parameters

pMsg

[in] Pointer to the **MSG** structure to be processed.

Return Values

S_OK

The page site processed the message.

S_FALSE

The page site did not process the message.

E_NOTIMPL

The page site does not support keyboard processing.

IPropertyPageSite - Ole Implementation

The system provides an implementation of the **IPropertyPageSite** interface through the **OleCreatePropertyFrame** or **OleCreatePropertyFrameIndirect** functions.

The current frame implementation provided through **OleCreatePropertyFrame** and **OleCreatePropertyFrameIndirect** only implements the **OnStatusChange** and **GetLocaleID** methods.

Remarks

OnStatusChange

Indicates that the property page has changed.

GetLocaleID

Returns the locale identifier so the property page can adjust itself to country-specific settings.

GetPageContainer

Returns E_NOTIMPL.

TranslateAccelerator

Returns E_NOTIMPL.

See Also

IPropertyPageSite

IPropertySetStorage Quick Info

Creates, opens, deletes, and enumerates property set storages that support instances of the **IPropertyStorage** interface. The **IPropertyStorage** interface manages a single property set in a property storage subobject; the **IPropertySetStorage** interface manages the storage of groups of such property sets. **IPropertySetStorage** can be supported by any file system entity, and is currently implemented in the OLE compound file object.

The **IPropertySetStorage** and **IPropertyStorage** interfaces provide a uniform way to create and manage property sets, whether or not these sets reside in a storage object that supports **IStorage**. When called through an object supporting **IStorage** (such as structured and compound files and directories) or **IStream**, the property sets created conform to the OLE property set format, described in detail in Appendix C of the OLE Programming Guide. Similarly, properties written using **IStorage** to the OLE property set format are visible through **IPropertySetStorage** and **IPropertyStorage**. **IPropertyStorage** does not support extensions to the OLE serialized property set format or multiple sections, because you can get equivalent functionality as simply by creating new sets or by adding new properties to existing property sets.

IPropertySetStorage methods identify property sets through a GUID called a format identifier (FMTID). The FMTID for a property set identifies the set of property identifiers in the property set, their meaning, and any constraints on the values. The format identifier of a property set should also provide the means to manipulate that property set. Only one instance of a given FMTID may exist at a time within a single property storage.

When to Implement

Implement **IPropertySetStorage** to store persistent properties in the file system. If you are using the OLE compound files implementation, you can use the implementation on the compound file object created through a call to **StgCreateDocfile** or **StgOpenStorage**. Once you have a pointer to any of the interface implementations (such as **IStorage**) on this object, you can call **QueryInterface** to get a pointer to the **IPropertySetStorage** interface implementation.

When to Use

Call **IPropertySetStorage** methods to create, open, or delete one or more property sets, or to enumerate the property sets contained in this property set storage.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPropertySetStorage Methods	Description
Create	Creates a new property set.
Open	Opens a previously created property set.
Delete	Deletes an existing property set.
Enum	Creates and retrieves a pointer to an object that can be used to enumerate property sets.

See Also

IPropertyStorage, IEnumSTATPROPSETSTG, STATPROPSETSTG, PROPVARIANT

IPropertySetStorage::Create Quick Info

Creates and opens a new property set in the property set storage object.

```
HRESULT Create(
  REFFMTID   fmtid,                //Format identifier of the property set to be created
  CLSID * pclsid,                  //Pointer to initial CLSID for this property set
  DWORD grfFlags,                  //PROPSETFLAG values
  DWORD grfMode,                   //Storage mode of new property set
  IPropertyStorage**  ppPropStg    //Indirect pointer to property storage sub-object
);
```

Parameters

fmtid

[in] Format identifier of the property set to be created.

pclsid

[in] Pointer to the initial CLSID for this property set. May be NULL, in which case it is set to all zeroes.

grfFlags

[in] Values from the PROPSETFLAG enumeration.

grfMode

[in] Access mode in which the newly created property set is to be opened, taken from certain values of the **STGM** enumeration, as described in the Remarks.

ppPropStg

[out] Indirect pointer to the **IPropertyStorage** interface on the new property storage sub-object.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The property set was created.

STG_E_FILEALREADYEXISTS

A property set of the indicated name already exists, and STGM_CREATE was not specified.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

STG_E_INVALIDPARAMETER

A parameter is invalid.

Remarks

IPropertySetStorage::Create creates and opens a new property set sub-object (supporting the **IPropertyStorage** interface) contained in this property set storage object. The property set automatically contains code page and locale ID properties. These are set to the current system default, and the current user default, respectively.

The *grfFlags* parameter is a combination of values taken from the enumeration **PROPSETFLAG**.

The *grfMode* parameter specifies the access mode in which the newly created set is to be opened. Values for this parameter are as in the like-named parameter to **IPropertySetStorage::Open,** with the addition of the following values:

Value	Meaning
STGM_FAILIFTHERE	If another property set with the specified *fmtid* already exists, the call fails. This is the default action; that is, unless STGM_CREATE is specified, STGM_FAILIFTHERE is implied.
STGM_CREATE	If another property set with the specified *fmtid* already exists, it is removed and replaced with this new one.
STGM_DIRECT	Open the property set without an additional level of transaction nesting. This is the default (the behavior if neither STGM_DIRECT nor STGM_TRANSACTED is specified).
STGM_TRANSACTED	Open the property set with an additional level of transaction nesting (beyond the transaction, if any, on this property set storage). This is possible only when you specify PROPSETFLAG_NONSIMPLE in the *grfFlags* parameter. Changes in the property set must be committed with **IPropertyStorage::Commit** before they are visible to the transaction on this property set storage.
STGM_READ	Read access is desired on the property set. Read permission is required on the property set storage.
STGM_WRITE	Write access is desired on the property set. Write permission is not required on the property set storage; however, such write permission is required for changes in the storage to be committed.
STGM_READWRITE	Read-write access is desired on the property set. Note that this flag is not the binary OR of the values STGM_READ and STGM_WRITE.
STGM_SHARE_EXCLUSIVE	Prevents others from subsequently opening the property set either in STGM_READ or STGM_WRITE mode.

Note The only access mode supported by **Create** is STGM_SHARE_EXCLUSIVE. To use the

resulting property set in an access mode other than STGM_SHARE_EXCLUSIVE, the caller should close the stream and then re-open it with a call to **IPropertySetStorage::Open**.

See Also

IPropertySetStorage::Open

IPropertySetStorage::Delete Quick Info

Deletes one of the property sets contained in the property set storage object.

```
HRESULT Delete(
  REFFMTID fmtid      //Format identifier of the property set to be deleted.
);
```

Parameters

fmtid

[in] Format identifier of the property set to be deleted.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The property set was successfully deleted.

STG_E_FILENOTFOUND

The specified property set does not exist.

STG_E_ACCESSDENIED

The requested access to the property set storage object has been denied.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

STG_E_INVALIDPARAMETER

The parameter is invalid.

Remarks

IPropertySetStorage::Delete deletes the property set specified by its format identifier. Specifying a property set that does not exist returns an error. Open substorages and streams (opened through one of the storage- or stream-valued properties) are put into the reverted state.

IPropertySetStorage::Enum Quick Info

Creates an enumerator object which contains information on the property sets stored in this property set storage. On return, this method supplies a pointer to the **IEnumSTATPROPSETSTG** pointer on the enumerator object.

```
HRESULT Enum(
  IEnumSTATPROPSETSTG**ppenum      //Indirect pointer to the new enumerator
);
```

Parameters

ppenum

[out] Indirect pointer to the **IEnumSTATPROPSETSTG** on the newly created enumeration object.

Return Values

S_OK

The enumerator object was successfully created.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

Remarks

IPropertySetStorage::Enum creates an enumerator object that can be used to iterate through STATPROPSETSTG structures. These sometimes provide information on the property sets managed by **IPropertySetStorage**. This method, on return, supplies a pointer to the **IEnumSTATPROPSETSTG** interface on this enumerator object on return.

See Also

IEnumSTATPROPSETSTG, **IEnumSTATPROPSETSTG -- Compound File Implementation**

IPropertySetStorage::Open Quick Info

Opens a property set contained in the property set storage object.

```
HRESULT Open(
  REFFMTID   fmtid,          //The format identifier of the property set to be opened
  DWORD grfMode,             //Storage mode in which property set is to be opened
  IPropertyStorage**  ppPropStg   //Indirect pointer to property storage object
);
```

Parameters

fmtid

[in] Format identifier of the property set to be opened.

grfMode

[in] Access mode in which the newly created property set is to be opened. These flags are taken from the STGM enumeration. Flags that may be used and their meanings in the context of this method are described in the Remarks.

ppPropStg

[in] Indirect pointer to the **IPropertyStorage** interface on the requested property storage sub-object.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

Success.

STG_E_FILENOTFOUND

A property set of the indicated name does not exist.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied, or the property set is corrupted.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

STG_E_INVALIDPARAMETER

A parameter is invalid.

Remarks

The mode in which the property set is to be opened is specified in the parameter *grfMode*. These flags are taken from the **STGM** enumeration, but, for this method, legal values and their meanings are as follows (only certain combinations of these flag values are legal).

Value	Meaning

STGM_DIRECT	Open the property set without an additional level of transaction nesting. This is the default (the behavior if neither STGM_DIRECT nor STGM_TRANSACTED is specified).
STGM_TRANSACTED	Open the property set with an additional level of transaction nesting (beyond the transaction, if any, on this property set storage object). Transacted mode is available only on non-simple property sets, because they use an **IStorage** with a contents stream. Changes in the property set must be committed with a call to **IPropertyStorage::Commit** before they are visible to the transaction on this property set storage.
STGM_READ	Open the property set with read access. Read permission is required on the property set storage.
STGM_WRITE	Open the property set with write access. Write permission is not required on the IPropertySetStorage; however, such write permission is required for changes in the storage to be committed.
STGM_READWRITE	Open the property set with read-write access. Note that this flag is not the binary OR of the values STGM_READ and STGM_WRITE.
STGM_SHARE_DENY_NONE	Subsequent openings of the property set are not denied read or write access. Not available in compound file implementation.
STGM_SHARE_DENY_READ	Subsequent openings of the property set in are denied read access. Not available in compound file implementation.
STGM_SHARE_DENY_WRITE	Subsequent openings of the property set are denied write access. This value is typically used to prevent making unnecessary copies of an object opened by multiple users. If this value is not specified, a snapshot is made, whether there are subsequent openings or not. Thus, you can improve performance by specifying this value. Not available in compound file implementation.
STGM_SHARE_EXCLUSIVE	The combination of STGM_SHARE_DENY_READ and STGM_SHARE_DENY_WRITE.

IPropertySetStorage-Compound File Implementation

The OLE compound file storage object implementation includes an implementation of both **IPropertyStorage**, the interface that manages a single persistent property set, and **IPropertySetStorage**, the interface that manages groups of persistent property sets.

To get a pointer to the compound file implementation of **IPropertySetStorage**, first call **StgCreateDocfile** to create a new compound file object or **StgOpenStorage** to open a previously created compound file. Both functions supply a pointer to the object's **IStorage** interface. When you want to deal with persistent property sets, call **IStorage::QueryInterface** for the **IPropertySetStorage** interface, specifying the header-defined name for the interface identifier **IID_IPropertySetStorage**.

When to Use

Call the methods of **IPropertySetStorage** to create, open, or delete property sets in the current compound file property set storage. There is also a method that supplies a pointer to an enumerator that can be used to enumerate the property sets in the storage.

Remarks

IPropertySetStorage::Create

Creates a new property set in the current compound file storage and, on return, supplies an indirect pointer to the **IPropertyStorage** compound file implementation. In this implementation, property sets may be transacted only if PROPSETFLAG_NONSIMPLE is specified.

IPropertySetStorage::Open

Opens an existing property set in the current property storage. On return, it supplies an indirect pointer to the compound file implementation of **IPropertyStorage**.

IPropertySetStorage::Delete

Deletes a property set in this property storage.

IPropertySetStorage::Enum

Creates an object that can be used to enumerate STATPROPSETSTG structures. Each STATPROPSETSTG structure provides information about a single property set. The implementation calls the constructor for IEnumSTATPROPSETSTG, which, in turn, uses the pointer to the **IStorage** interface to create a **STATSTG** enumerator, which is then used over the actual storage to get the information about the property sets.

Note The DocumentSummaryInformation property set is special, in that it may have two property set sections. This property set is described in the OLE Programmer's Reference, in the section titled The DocumentSummaryInformation Property Set. The second section is referred to as the User-Defined Properties. Each section is identified with a unique Format ID, for example FMTID_DocumentSummaryInformation and FMTID_UserDefinedProperties.

When **IPropertySetStorage::Create** is called to create the User-Defined Property Set, the first section is created automatically. Thus once FMTID_UserDefinedProperties is created, FMTID_DocumentSummaryInformation need not be created, but can be opened with a call to **IPropertySetStorage::Open**. Note that creating the first section does not automatically create the second section. It is not possible to open both sections simultaneously.

When **IPropertySetStorage::Create** is called to create the User-Defined Property Set, the first section is created automatically. Thus once FMTID_UserDefinedProperties is created, FMTID_DocumentSummaryInformation need not be created, but can be opened with a call to **IPropertySetStorage::Open**. Note that creating the first section does not automatically create the second section. It is not possible to open both sections simultaneously.

Alternately, when **IPropertySetStorage::Delete** is called to delete the **first** section, both sections are deleted. That is, calling **IPropertySetStorage::Delete** with FMTID_DocumentSummaryInformation, causes both that section and the FMTID_UserDefinedProperties section to be deleted. Note that deleting the second section does not automatically delete the first section.

Finally, when **IPropertySetStorage::Enum** is used to enumerate property sets, the User-Defined Property Set will not be enumerated.

See Also

IPropertyStorage, **IPropertySetStorage - Compound File Implementation**, **STATPROPSETSTG structure**, **PROPSETFLAG** enumeration, **IStorage::EnumElements**

IPropertyStorage Quick Info

Manages the persistent properties of a single property set. Persistent properties consist of information that can be stored persistently in a property set, such as the summary information associated with a file. This contrasts with run-time properties associated with Controls and Automation, which can be used to affect system behavior. Use the methods of the **IPropertySetStorage** interface to create or open a persistent property set. An **IPropertySetStorage** instance can manage zero or more **IPropertyStorage** instances.

Each property within a property set is identified by a property identifier, a four-byte ULONG value unique to that set. You can also assign a string name to a property through the **IPropertyStorage** interface.

Property identifiers are different from the dispatch identifiers used in Automation *dispid* property name tags. One difference is that the general-purpose use of property identifier values zero and one is prohibited in **IPropertyStorage**, while no such restriction exists in **IDispatch**. In addition, while there is significant overlap in the data types for property values that may be used in **IPropertyStorage** and **IDispatch**, the sets are not identical. Persistent property data types used in **IPropertyStorage** methods are defined in the **PROPVARIANT** structure.

When to Implement

Implement **IPropertyStorage** when you want to store properties in the file system. If you are using the OLE compound files implementation, the compound file object created through a call to **StgCreateDocfile** includes an implementation of **IPropertySetStorage**, which allows access to the implementation of **IPropertyStorage**. Once you have a pointer to any of the interface implementations (such as **IStorage**) on this object, you can call **QueryInterface** to get a pointer to the **IPropertySetStorage** interface implementation, and then call either the **Open** or **Create** method, as appropriate to obtain a pointer to the **IPropertyStorage** interface managing the specified property set.

When to Use

Use IPropertyStorage to create and manage properties that are stored in a given property set.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IPropertyStorage Methods	Description
ReadMultiple	Reads property values in a property set.
WriteMultiple	Writes property values in a property set.
DeleteMultiple	Deletes properties in a property set.
ReadPropertyNames	Gets corresponding string names for given property identifiers.
WritePropertyNames	Creates or changes string names corresponding to given property identifiers.
DeletePropertyNames	Deletes string names for given property identifiers.

SetClass	Assigns a CLSID to the property set.
Commit	As in **IStorage::Commit**, flushes or commits changes to the property storage object.
Revert	When the property storage is opened in transacted mode, discards all changes since the last commit.
Enum	Creates and gets a pointer to an enumerator for properties within this set.
Stat	Receives statistics about this property set.
SetTimes	Sets modification, creation, and access times for the property set.

See Also

IPropertySetStorage, **IEnumSTATPROPSTG**, **IEnumSTATPROPSETSTG**, **STATPROPSTG**, **STATPROPSETSTG**, **PROPVARIANT**

IPropertyStorage::Commit Quick Info

Saves any changes made to a property storage object to the parent storage object.

```
HRESULT Commit(
  DWORD   grfCommitFlags     //Flags specifying conditions for the commit
);
```

Parameters

grfCommitFlags

[in] Flags specifying the conditions under which the commit is to be performed. Specific flags and their meanings are described in the following Remarks section.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The changes were saved successfully.

STG_E_NOTCURRENT

STGC_ONLYIFCURRENT was specified, but the optimistic concurrency control failed.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

STG_E_INVALIDPARAMETER

One or more flags specified in *grfCommitFlags* is invalid.

Remarks

As in **IStorage::Commit**, ensures that any changes made to a property storage object are reflected in the parent storage.

In direct mode in the compound file implementation, this call causes any changes currently buffered up in memory to be flushed to the underlying property stream. In the compound file implementation for non-simple property sets, **IStorage::Commit** is also called on the underlying substorage object with the passed *grfCommitFlags* parameter.

In transacted mode, this method causes the changes to be permanently reflected in the persistent image of the storage object. The changes that are committed must have been made to this property set since it was opened or since the last commit on this opening of the property set. One could think of the action of committing as publishing the changes that this level currently knows about one more layer outwards. Of course, this is still subject to any outer level transaction that may be present on the object in which this property set is contained. Write permission must be specified when the property set is opened (through **IPropertySetStorage**) on the property set opening for the commit operation to succeed.

If the commit operation fails for any reason, the state of the property storage object is as it was before the commit.

This call has no effect on existing storage- or stream-valued properties opened from this property storage, but it does commit them.

Valid values for the *grfCommitFlags* parameter are as follows:

Value	Meaning
STGC_DEFAULT	Commit per the usual transaction semantics. Last writer wins. This flag may not be specified with other flag values.
STGC_ONLYIFCURRENT	Commit the changes only if the current persistent contents of the property set are the ones on which the changes about to be committed are based. That is, do not commit changes if the contents of the property set have been changed by a commit from another opening of the property set. The error STG_E_NOTCURRENT is returned if the commit does not succeed for this reason.
STGC_OVERWRITE	Only useful when committing a transaction which has no further outer nesting level of transactioning, though legal in all cases. Indicates that the caller is willing to take some risk of data corruption at the expense of a decreased usage of disk on the destination volume. This flag is potentially useful in low disk space scenarios, though should be used only with caution.

See Also

IPropertyStorage::ReadMultiple, **IStorage::Commit**

IPropertyStorage::DeleteMultiple Quick Info

Deletes as many of the indicated properties as exist in this property set.

```
HRESULT DeleteMultiple(
    ULONG cpspec,              //Count of properties to be deleted
    PROPSPEC const             //Array of properties to be deleted
rgpspec[]
);
```

Parameters

cpspec

[in] Count of properties being deleted. May legally be zero, though this is a no-op, deleting no properties.

rgpspec[]

[in] Properties to be deleted. A mixture of property identifiers and string-named properties is permitted. There may be duplicates, and there is no requirement that properties be specified in any order.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

All of the specified properties that exist in the property set have been deleted.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied. No properties were deleted.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation. Some properties may not have been deleted.

STG_E_INVALIDPARAMETER

At least one of the parameters is invalid, as when one of the PROPSPECs contains an illegal *ulKind* value. Some properties may not have been deleted.

STG_E_INVALIDPOINTER

May be returned when at least one of the pointers passed in is invalid. Some properties may not have been written. More frequently, an invalid pointer will instead result in an access violation.

Remarks

IPropertyStorage::DeleteMultiple must delete as many of the indicated properties as are in the current property set. If a deletion of a stream- or storage-valued property occurs while that property is open, the deletion will succeed and place the previously returned **IStream** or **IStorage** pointer in the reverted state.

IPropertyStorage::DeletePropertyNames Quick Info

Deletes specified string names from the current property set.

```
HRESULT DeletePropertyNames(
    ULONG cpropid,          //Size of the rgpropid array
    PROPID const            //Property identifiers for which string names are to be deleted
rgpropid[]
);
```

Parameters

cpropid

[in] The size on input of the array *rgpropid*. If 0, no property names are deleted.

rgpropid[]

[in] Property identifiers for which string names are to be deleted.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

Success. The names of all of the indicated properties that exist in this set have been deleted.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied. No property names were deleted.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation. Some property names may not have been deleted.

STG_E_INVALIDPARAMETER

At least one of the parameters is invalid. Some property names may not have been deleted.

Remarks

For each property identifier in *rgpropid*, **IPropertyStorage::DeletePropertyNames** removes the corresponding name-to-property identifier mapping, if any. An attempt to delete the name of a property that either does not exist or does not presently have a string name associated with it is silently ignored. This method has no effect on the properties themselves.

Note All the stored string property names can be deleted by deleting property identifier zero, but *cpropid* must be equal to 1 for this to not be an invalid parameter error.

See Also

IPropertyStorage::ReadPropertyNames

IPropertyStorage::Enum Quick Info

Creates an enumerator object designed to enumerate data of type STATPROPSTG, which contains information on the current property set. On return, this method supplies a pointer to the **IEnumSTATPROPSTG** pointer on this object.

```
HRESULT Enum(
    IEnumSTATPROPSTG**          //Indirect pointer to new enumerator
ppenum
);
```

Parameters

ppenum

[out] Indirect pointer to the **IEnumSTATPROPSTG** interface on the new enumeration object.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

A pointer to the enumerator has been retrieved.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

STG_E_INVALIDPARAMETER

The parameter is invalid.

STG_E_READFAULT

Error reading storage.

Remarks

IPropertyStorage::Enum creates an enumeration object that can be used to iterate STATPROPSTG structures. On return, this method supplies a pointer to an instance of **IEnumSTATPROPSTG** interface on this objects whose methods you can call to obtain information on the current property set.

See Also

IEnumSTATPROPSTG, **IEnumSTATPROPSTG -- Compound File Implementation**

IPropertyStorage::ReadMultiple Quick Info

Reads specified properties from the current property set.

```
HRESULT ReadMultiple(
    ULONG cpspec,                   //Count of properties being read.
    PROPSPEC const                  //Array of the properties to be read
rgpspec[],
    PROPVARIANT rgvar[]             //Array of PROPVARIANTs containing the property values on
                                    return
);
```

Parameters

cpspec

[in] Count of properties specified in the *rgpspec* array. May legally be zero, though this is a no-op, reading no properties.

rgpspec[]

[in] The properties to be read in the **PROPSPEC** structures. Properties can be specified either by property identifier or by optional string name. It is not necessary to specify properties in any particular order in the array. The array can contain duplicate properties, resulting in duplicate property values on return for simple properties. Non-simple properties should return access denied on an attempt to open them a second time. The array can contain a mixture of property identifiers and string identifiers.

rgvar[]

[in, out] Caller-allocated array of PROPVARIANTs that, on return, contains the values of the properties specified by *rgpspec*. The array must be able to receive at least *cpspec* PROPVARIANTs. The caller does not need to initialize these PROPVARIANTs in any particular way; the implementation must fill in all field members correctly on return. If there is no other appropriate value, the implementation must set the *vt* member of each **PROPVARIANT** to VT_EMPTY.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

Success. At least some of the requested properties were retrieved.

S_FALSE

All the property names or identifiers had valid syntax, but none of them exist in this property set. Accordingly, no properties were retrieved., and each **PROPVARIANT** structure is set to VT_EMPTY.

STG_E_ACCESSDENIED

The requested access to the property set has been denied, or, when one or more of the properties is a stream or storage object, access to that substorage or substream has been denied. (The storage or stream may already be open). No properties were retrieved.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation. No properties were retrieved.

STG_E_INVALIDPARAMETER

At least one of the parameters is invalid, such as when one of the PROPSPECs contains an illegal *ulKind* value. No properties were retrieved.

STG_E_INVALIDPOINTER

At least one of the pointers passed in is invalid. No properties were retrieved.

HRESULT_FROM_WIN32(ERROR_NO_UNICODE_TRANSLATION)

There was a failed attempt to translate a Unicode string to or from Ansi.

Remarks

IPropertyStorage::ReadMultiple reads as many of the properties specified in the *rgpspec* array as are found in the property set. As long as any of the properties requested is read, a request to retrieve a property that does not exist is not an error. Instead, this must cause VT_EMPTY to be written for that property to the *rgvar*[] array on return. When none of the requested properties exist, the method should return S_FALSE, and set VT_EMPTY in each PROPVARIANT. If any other error is returned, no property values are retrieved, and the caller need not worry about releasing them.

The *rgpspec* parameter is an array of **PROPSPEC** structures, which specify for each property either its property identifier or, if one is assigned, a string identifier. You can map a string to a property identifier by calling **IPropertyStorage::WritePropertyNames**. The use of property identifiers is, however, likely to be significantly more efficient than the use of strings.

Properties that are requested by string name (PRSPEC_LPWSTR) are mapped case-insensitively to property identifiers as they are specified in the current property set (and according to the current system locale).

All propvariants, except for those that are pointers to streams and storages, are called simple propvariants. These simple propvariants receive data by value, so a call to **IPropertyStorage::ReadMultiple** supplies a copy of the data that the caller then owns. To create or update these properties, call **IPropertyStorage::WriteMultiple**.

In contrast, the variant types VT_STREAM, VT_STREAMEDOBJECT, VT_STORAGE, and VT_STOREDOBJECT are non-simple properties, because rather than supplying a value, the method retrieves a pointer to the indicated interface, from which the data can then be read. These types permit the storage of large amounts of information through a single property. There are several issues that arise in using non-simple properties.

To create these properties, as for the other properties, call **IPropertyStorage::WriteMultiple**. Rather than calling the same method to update, however, it is more efficient to first call **IPropertyStorage::ReadMultiple** to get the interface pointer to the stream or storage, then write data using the **IStream** or **IStorage** methods. A stream or storage opened through a property is always opened in direct mode, so an additional level of nested transaction is not introduced. There may, however, still be a transaction on the property set as a whole, depending on how it was opened or created through **IPropertySetStorage**. Further, the access and share mode tags specified when the property set is opened or created, are passed to property-based streams or storages.

The lifetimes of property-based stream or storage pointers, although theoretically independent of their associated **IPropertyStorage** and **IPropertySetStorage** pointers, in fact, effectively depend on them. The data visible through the stream or storage is related to the transaction on the property storage object from which it is retrieved, just as for a storage object (supporting **IStorage**) with contained stream and storage sub-objects. If the transaction on the parent object is aborted, existing **IStream** and **IStorage** pointers subordinate to that object enter a "zombie" state. Because **IPropertyStorage** is the only interface on the property storage object, the useful lifetime of the contained **IStream** and **IStorage** pointers is bounded by the lifetime of the **IPropertyStorage** interface.

The implementation must also deal with the situation where the same stream- or storage-valued property is requested multiple times through the same **IPropertyStorage** interface instance. For example, in the OLE compound file implementation, the open will succeed or fail depending on whether or not the property is already open.

Another issue is multiple opens in transacted mode. The result depends on the isolation level that was specified through a call to **IPropertySetStorage** methods, (either the **Open** or **Create** method, through the STGM flags) at the time that the property storage was opened .

If the call to open the property set specifies read-write access, **IStorage**- and **IStream**-valued properties are always opened with read-write access. Data can then be written through these interfaces, changing the value of the property, which is the most efficient way to update these properties. The property value itself does not have an additional level of transaction nesting, so changes are scoped under the transaction (if any) on the property storage object.

See Also

IPropertySetStorage, **IPropertyStorage::WriteMultiple**, **IPropertyStorage::WritePropertyNames**

IPropertyStorage::ReadPropertyNames Quick Info

Retrieves any existing string names for the specified property identifiers.

```
HRESULT ReadPropertyNames(
  ULONG   cpropid,          //Number of elements in rgpropid
  PROPID const  rgpropid[],  //Property identifiers for which names are to be retrieved.
  LPWSTR   rglpwstrName[]   //Array of returned string names
);
```

Parameters

cpropid

[in] Number of elements on input of the array *rgpropid*. May legally be zero, though this is a no-op, reading no property names.

rgpropid[]

[in] Array of property identifiers for which names are to be retrieved.

rglpwstrName[]

[in, out] Caller-allocated array of size cpropid of LPWSTRs. On return, the implementation fills in this array. A given entry contains either the corresponding string name of a property identifier or NULL if the property identifier has no string name.

Each LPWSTR member of the array should be freed using **CoTaskMemFree**.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

One or more string names were retrieved and all members of *rglpwstrName* are valid (either NULL or a valid LPWSTR).

S_FALSE

No string names were retrieved because none of the requested property identifiers have string names presently associated with them in this property storage object (this result does not address whether the given property identifiers presently exist in the set).

STG_E_INVALIDHEADER

The property name dictionary was not found.

STG_E_READFAULT

Error reading the storage.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied. No string names were retrieved.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation. No string names were retrieved.

STG_E_INVALIDPARAMETER

A parameter is invalid. No string names were retrieved.

HRESULT_FROM_WIN32(ERROR_NO_UNICODE_TRANSLATION)

There was a failed attempt to translate a Unicode string to or from Ansi.

Remarks

For each property identifier in the list of property identifiers supplied in the *rgpropid* array, **IPropertyStorage::ReadPropertyNames** retrieves the corresponding string name, if there is one. String names are created either by specifying the names in calls to **IPropertyStorage::WriteMultiple** when you are creating the property, or through a call to **IPropertyStorage::WritePropertyNames**. In any case, the string name is optional; all properties must have a property identifier.

String names mapped to property identifiers must be unique within the set.

See Also

IPropertyStorage::WritePropertyNames, **IPropertyStorage::WriteMultiple**

IPropertyStorage::Revert Quick Info

Discards all changes to the property set it was opened or changes were last committed. Has no effect on a direct-mode property set.

HRESULT Revert();

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

Success.

Remarks

For transacted-mode property sets, discards all changes that have been made in this property set since set was opened or the time it was last committed (depending on which is later). After this operation, any existing storage- or stream-valued properties that have been opened from the property set being reverted are invalid and can no longer be used. The error STG_E_REVERTED will be returned on all calls except **Release** using these streams or storages.

For direct-mode property sets, this request is ignored and returns S_OK.

See Also

IPropertyStorage::Commit

IPropertyStorage::Stat Quick Info

Retrieves information about the current open property set.

```
HRESULT Stat(
  STATPROPSTG* pstatpsstg     //Pointer to a filled-in STATPROPSETSTG structure
);
```

Parameters

pstatpsstg

[out] Pointer to a STATPROPSETSTG structure, which contains statistics about the current open property set.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

Statistics were successfully obtained.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

STG_E_INVALIDPARAMETER

The parameter is invalid.

Remarks

IPropertyStorage::Stat fills in and returns a pointer to a **STATPROPSETSTG** structure, containing statistics about the current property set. STATPROPSETSTG fields have the following meanings:

Field	Meaning
fmtid	The FMTID of this property set, specified when the property set was initially created.
clsid	The CLSID of this property set, specified when the property set was initially created and possibly modified thereafter with **IPropertyStorage::SetClass**. If not set, the value will be CLSID_NULL.
grfFlags	The flag values this set was created with. For details, see **IPropertySetStorage::Create**.
mtime	The time in UTC (FILETIME) at which this property set was last modified. Not all **IPropertyStorage** implementations maintain modification times on property sets; those who do not will return zero for this value.
ctime	The time in UTC (FILETIME) at which this property set

	was created. Not all IPropertyStorage implementations maintain creation times on property sets; those that do not will set this value to 0.
atime	The time in UTC (FILETIME) at which this property set was last accessed. Not all IPropertyStorage implementations maintain last access times on property sets; those that do not will set this value to 0.

See Also

STATPROPSETSTG structure, **IPropertySetStorage::Enum**, FILETIME structure

IPropertyStorage::SetClass Quick Info

Assigns a new CLSID to the current property storage object, and persistently stores the CLSID with the object.

```
HRESULT SetClass(
  REFCLSID   clsid     //New CLSID for the property set
);
```

Parameters

clsid

[in] New CLSID to be associated with the property set.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The CLSID has been assigned.

STG_E_ACCESSDENIED

The requested access to the **IPropertyStorage** interface has been denied. The CLSID was not assigned.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation. The CLSID was not assigned.

STG_E_INVALIDPARAMETER

The parameter is invalid. The CLSID was not assigned.

Remarks

Assigns a CLSID to the current property storage object. The CLSID has no relationship to the stored property identifiers. Assigning a CLSID allows a piece of code to be associated with a given instance of a property set; such code, for example, might manage the user interface. Different CLSIDs can be associated with different property set instances that have the same FMTID.

If the property set is created with NULL specified as the **IPropertySetStorage::Create** *pclsid* parameter, the CLSID is set to all zeroes.

The current CLSID on a property storage object can be retrieved with a call to **IPropertyStorage::Stat**. The initial value for the CLSID can be specified at the time that the storage is created with a call to **IPropertySetStorage::Create**.

Setting the CLSID on a non-simple property set (one that can legally contain storage- or stream-valued properties, as described in **IPropertySetStorage::Create**) also sets the CLSID on the underlying sub-storage.

See Also

IPropertySetStorage::Create, **IPropertyStorage::Stat**

IPropertyStorage::SetTimes Quick Info

Sets the modification, access, and creation times of this property set, if supported by the implementation. Not all implementations support all these time values.

```
HRESULT SetTimes(
    FILETIME const *   pctime,      //New creation time for the property set
    FILETIME const *   patime,      //New access time for the property set
    FILETIME const *   pmtime       //New modification time for the property set
);
```

Parameters

pctime

[in] Pointer to the new creation time for the property set. May be NULL, indicating that this time is not to be modified by this call.

patime

[in] Pointer to the new access time for the property set. May be NULL, indicating that this time is not to be modified by this call.

pmtime

[in] Pointer to the new modification time for the property set. May be NULL, indicating that this time is not to be modified by this call.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

All the requested times have been successfully updated.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied; no times have been updated.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation.

STG_E_INVALIDPARAMETER

The parameter is invalid. This error is returned if an attempt is made to set a time value which is not supported by this implementation.

Remarks

Sets the modification, access, and creation times of the current open property set, if supported by the implementation (not all implementations support all these time values). Unsupported timestamps are always reported as zero, enabling the caller to test for support. A call to **IPropertyStorage::Stat** supplies (among other information) timestamp information.

Notice that this functionality is provided as an **IPropertyStorage** method on a property storage object that is already open, in contrast to being provided as a method in **IPropertySetStorage**. Normally, when the

SetTimes method is not explicitly called, the access and modification times are updated as a side effect of reading and writing the property set. When **SetTimes** is used, the latest specified times supersede either default times or time values specified in previous calls to **SetTimes**.

See Also

IPropertyStorage::Stat, FILETIME structure

IPropertyStorage::WriteMultiple Quick Info

Writes a specified group of properties to the current property set. If a property with a specified name already exists, it is replaced, even when the old and new types for the property value are different. If a property of a given name or property identifier does not exist, it is created.

```
HRESULT WriteMultiple(
    ULONG cpspec,                   //The number of properties being set.
    PROPSPEC const rgpspec[],       //Property specifiers
    PROPVARIANT const rgvar[],      //Array of PROPVARIANT values
    PROPID propidNameFirst          //Minimum value for property identifiers when they must be allocated
);
```

Parameters

cpspec

[in] The number of properties being set. May legally be zero, though this is a no-op, writing no properties.

rgpspec[]

[in] Array of the specifiers to which properties are to be set. These are in no particular order, and may legally contain duplicates (the last specified is to take effect). A mixture of property identifiers and string names is permitted.

rgvar[]

[in] An array (of size *cpspec*) of PROPVARIANTs that contain the property values to be written. The array must be of the size specified by *cpspec*.

propidNameFirst

[in] Specifies the minimum value for the property identifiers the method must assign if the *rgpspec* parameter specifies string-named properties for which no property identifiers currently exist. If all string-named properties specified already exist in this set, and thus already have property identifiers, this value is ignored. When not ignored, this value must be at least two (property identifiers 0and 1 are reserved for special uses) and less than 0x80000000 (property identifier values beyond that are reserved for special use).

HRESULT_FROM_WIN32(ERROR_NO_UNICODE_TRANSLATION)

There was a failed attempt to translate a Unicode string to or from Ansi.

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

All of the indicated properties were successfully written.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied. No properties have been

written. The property set was opened in STGM_READ mode.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation. Some properties may or may not have been written.

STG_E_INVALIDPARAMETER

At least one of the parameters is invalid. Some properties may not have been written. This error would be returned in several situations, for example: 1) *rgvar* may be NULL; 2) a stream- or storage-valued property is present in *rgpspec* but the property set was created without PROPSETFLAG_NONSIMPLE; 3) one or more property variant types may be invalid; 4) one of the PROPSPECs contains an illegal *ulKind* value.

STG_E_INVALIDPOINTER

May be returned when at least one of the pointers passed in is invalid. Some properties may or may not have been written. More frequently, an invalid pointer will instead result in an access violation.

STG_E_WRITEFAULT

Error writing the storage.

STG_E_REVERTED

The property set was reverted. For example, if the property set is deleted while open (by using **IPropertySetStorage::Delete**) this status would be returned.

STG_E_MEDIUMFULL

The disk is full. Some properties may or may not have been written.

STG_E_PROPSETMISMATCHED

An attempt was made to write a non-simple (stream- or storage-valued) property to a simple property set.

Remarks

If a specified property already exists, it's value is replaced with the new one, even when the old and new types for the property value are different. If you specify a property identifier that does not exist, that property is created. If a string name is supplied for a property which does not exist, the method will allocate a property identifier for that property, and the name will be added to the dictionary.

When allocating a property identifier, the implementation can choose any value not currently in use in the property set for a property identifier, as long as it is not 0 or 1 or greater than 0x80000000, all of which are reserved values. The *propidNameFirst* parameter establishes a minimum value for property identifiers within the set, and must be greater than 1 and less than 0x80000000.

If there is an attempt to write a property that already exists with an invalid parameter, the method should return STG_E_INVALIDPARAMETER; if the property does not exist, it should not be created. This behavior facilitates the use of a **ReadMultiple** - update - **WriteMultiple** sequence to update a group of properties without requiring that the calling code ensure that all the requested properties in the call to **ReadMultiple** were retrieved.

It is recommended that property sets be created as Unicode, by not setting the PROPSETFLAG_ANSI flag in the *grfFlags* parameter of **IPropertySetStorage::Create**. It is also recommended that you avoid using VT_LPSTR values, and use VT_LPWSTR values instead. When the property set code page is Unicode, VT_LPSTR string values are converted to Unicode when stored, and back to multibyte string values when retrieved. When the code page of the property set is not Unicode, property names, VT_BSTR strings, and non-simple property values are converted to multibyte strings when stored, and

converted back to Unicode when retrieved, all using the current system ANSI code page.

To create stream or storage object as a property in a nonsimple property set, call **IPropertyStorage::WriteMultiple**. While you would also call this method to update simple properties, it is not an efficient way to update stream and storage objects in a property set. This is because updating one of these properties through a call to **WriteMultiple** creates in the property storage object a copy of the passed-in data, and the **IStorage** or **IStream** pointers are not retained beyond the duration of this call. It is usually more efficient to update stream or storage objects by first calling **IPropertyStorage::ReadMultiple** to get the interface pointer to the stream or storage, then writing data through the **IStream** or **IStorage** methods.

A stream or storage opened through a property is always opened in direct mode, so an additional level of nested transaction is not introduced. There is still likely to be a transaction on the property set as a whole. Further, a property-based stream or storage is opened in read-write mode, if possible, given the mode on the property set; otherwise, read mode is used.

When the copy is made, the underlying CopyTo operation on VT_STREAM properties operates on the current seek position of the source. The seek position is destroyed on failure, but on success it is at EOF.

If a stream or storage property does not exist, passing an **IStream** or **IStorage** pointer with a value of NULL creates an empty stream or storage property value. If a stream or storage property is already open from a call to **ReadMultiple**, a NULL value must cause the **WriteMultiple** operation to truncate it and return S_OK, placing the previously returned stream- and storage-valued pointers into the reverted state (as happens in the compound file implementation.)

Storage- and stream-valued properties always manifest themselves to downlevel clients as sibling streams or storages to the stream containing the main contents of the property set—they are never stored directly in-line in the property set. This allows smooth interoperability and control when down-level clients interact with up-level clients. Thus, from a downlevel perspective, property sets containing **IStream** or **IStorage** valued properties are always stored in a storage object, not a stream. The specific name of the sibling used is completely under the control of the **IPropertyStorage** implementation, as long as the name is from the non-reserved part of the **IStorage** name space. See Appendix C of the OLE Programmer's Guide for a discussion of the serialized property set format for further details. As is described there, the string name is stored in the same format as a VT_BSTR. Refer also to the earlier discussion in this method of multibyte to Unicode conversions for property names.

If the **WriteMultiple** method returns an error when writing stream- or storage-valued properties (indirect properties), the amount of data actually written is undefined. If the caller requires consistency of the property set and its indirect properties when writing stream- and/or storage-valued properties, use of transacted mode is advised.

If an implicit deletion of a stream- or storage-valued property occurs while that property is open, (as, for example, when a VT_I4 is written over a VT_STREAM), the deletion will succeed and place the previously returned IStream pointer in the reverted state.

See Also

IPropertySetStorage::Create, **IPropertyStorage::ReadMultiple**

IPropertyStorage::WritePropertyNames

Assigns string names to a specified array of property IDs in the current property set.

```
HRESULT WritePropertyNames(
    ULONG    cpropid,          //Size on input of the array rgpropid
    PROPID const  rgpropid[],  //Property identifiers for which names are to be set
    LPWSTR const               //New names of the corresponding property identifiers
rglpwstrName[],
);
```

Parameters

cpropid

[in] Size on input of the array *rgpropid*. May legally be zero, though this is a no-op, writing no property names.

rgpropid[]

[in] Array of the property identifiers for which names are to be set.

rglpwstrName[]

[in] Array of new names to be assigned to the corresponding property identifiers in the *rgpropid* array. These names may not exceed 255 characters (not including the NULL terminator).

Return Values

This method supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

Success. All of the indicated string names were successfully set.

STG_E_INVALIDNAME

At least one of the indicated property identifier values does not exist in this property set. No names were set.

STG_E_ACCESSDENIED

The requested access to the property storage object has been denied. No property names have been changed in the storage.

STG_E_INSUFFICIENTMEMORY

There is not sufficient memory to perform this operation. Some names may not have been set.

STG_E_INVALIDPARAMETER

A parameter is invalid. Some names may not have been set.

HRESULT_FROM_WIN32(ERROR_NO_UNICODE_TRANSLATION)

There was a failed attempt to translate a Unicode string to or from Ansi.

Remarks

IPropertyStorage::WritePropertyNames assigns string names to property identifiers passed to the

method in the *rgpropid* array. It associates each string name in the *rglpwstrName* array with the respective property identifier in *rgpropid*. It is explicitly valid to define a name for a property identifier that is not currently present in the property storage object.

It is also valid to change the mapping for an existing string name (determined by a case-insensitive match). That is, you can use the **WritePropertyNames** method to map an existing name to a new property identifier, or to map a new name to a property identifier that already has a name in the dictionary. In either case, the original mapping is deleted. Property names must be unique (as are property identifiers) within the property set.

The storage of string property names preserves the case. String property names are limited in length to 128 characters. Property names that begin with the binary Unicode characters 0x0001 through 0x001F are reserved for future use.

See Also

IPropertyStorage::ReadPropertyNames, **IPropertyStorage::ReadMultiple**, **IPropertyStorage::WriteMultiple**

IPropertyStorage-Compound File Implementation

The OLE implementation of the Structured Storage architecture is called compound files. Storage objects as implemented in compound files include an implementation of both **IPropertyStorage**, the interface that manages a single persistent property set, and **IPropertySetStorage**, the interface that manages groups of persistent property sets.

To get a pointer to the compound file implementation of **IPropertyStorage**, first call **StgCreateDocfile** to create a new compound file object or **StgOpenStorage**, to open a previously created compound file. Both functions supply a pointer to the object's **IStorage** interface. When you want to deal with persistent property sets, call **QueryInterface** for the **IPropertySetStorage** interface, specifying the header-defined name for the interface identifier **IID_IPropertySetStorage**. Calling either the **Create** or **Open** method of that interface, you get a pointer to the **IPropertyStorage** interface, which you can use to call any of its methods.

When to Use

Use **IPropertyStorage** to manage properties within a single property set. Its methods support reading, writing, and deleting both properties and the optional string names that can be associated with property identifiers. Other methods support the standard commit and revert storage operations. There is also a method that allows you to set times associated with the property storage, and another that permits the assignment of a CLSID that can be used to associate other code, such as user interface code, with the property set. Calling the **Enum** method supplies a pointer to the compound file implementation of **IEnumSTATPROPSTG**, which allows you to enumerate the properties in the set.

Remarks

The compound file implementation of **IPropertyStorage** caches open property sets in memory in order to improve performance. As a result, changes to a property set are not written to the compound file until the **Commit** or **Release** (last reference) methods are called.

IPropertyStorage::ReadMultiple

Reads the properties specified in the *rgpspec* array and supplies the values of all valid properties in the *rgvar* array of PROPVARIANTs. In the OLE compound file implementation, duplicate property identifiers that refer to stream- or storage-types result in multiple calls to **IStorage::OpenStream** or **IStorage::OpenStorage** and the success or failure of **ReadMultiple** depends on the underlying storage implementation's ability to share opens. Because in a compound file STGM_SHARE_EXCLUSIVE is forced, multiple opens will fail. Opening the same storage object more than once from the same parent storage is not supported. The STGM_SHARE_EXCLUSIVE flag must be specified.

In addition, to ensure thread-safe operation if the same stream- or storage-valued property is requested multiple times through the same **IPropertyStorage** pointer in the OLE compound file implementation, the open will succeed or fail depending on whether or not the property is already open and on whether the underlying file system handles multiple opens of a stream or storage. Thus, the **ReadMultiple** operation on a stream- or storage-valued property always results in a call to **IStorage::OpenStream**, or **IStorage::OpenStorage**, passing the access (STGM_READWRITE, etc.) and share flags (STGM_SHARE_EXCLUSIVE, etc) specified when the original property set was opened or created.

If the method fails, the values written to *rgvar*[] are undefined. If some stream- or storage-valued properties are opened successfully but an error occurs before execution is complete, these should be released before the method returns.

IPropertyStorage::WriteMultiple

Writes the properties specified in the *rgpspec*[] array, assigning them the PROPVARIANT tags and values specified in *rgvar*[]. Properties that already exist are assigned the specified PROPVARIANT

values, and properties that do not currently exist are created.

IPropertyStorage::DeleteMultiple

Deletes the properties specified in the *rgpspec*[].

IPropertyStorage::ReadPropertyNames

Reads existing string names associated with the property identifiers specified in the *rgpropid*[] array.

IPropertyStorage::WritePropertyNames

Assigns string names specified in the *rglpwstrName* array to property identifiers specified in the *rgpropid* array.

IPropertyStorage::DeletePropertyNames

Deletes the string names of the property identifiers specified in the rgpropid array by writing NULL to the property name.

IPropertyStorage::SetClass

Sets the CLSID field of the property set stream. In this implementation, setting the CLSID on a non-simple property set (one that can legally contain storage- or stream-valued properties, as described in **IPropertySetStorage::Create**) also sets the CLSID on the underlying sub-storage so that it can be obtained through a call to **IStorage::Stat**.

IPropertyStorage::Commit

For both simple and non-simple property sets, flushes the memory image to the disk subsystem. In addition, for non-simple transacted-mode property sets, this method performs a commit (as in **IStorage::Commit**) on the property set.

IPropertyStorage::Revert

For non-simple property sets only, calls the underlying storage's **Revert** method and re-opens the 'contents' stream. For simple property sets, returns E_OK.

IPropertyStorage::Enum

Constructs an instance of **IEnumSTATPROPSTG**, the methods of which can be called to enumerate the STATPROPSTG structures that provide information about each of the properties in the set. This implementation creates an array into which the entire property set is read and which can be shared when **IEnumSTATPROPSTG::Clone** is called.

IPropertyStorage::Stat

Fills in the fields of a STATPROPSETSTG structure, which contains information about the property set as a whole. On return, supplies a pointer to the structure. For non-simple storage sets, this implementation calls **IStorage::Stat** (or **IStream::Stat**) to get the times from the underlying storage or stream. For simple storage sets, no times are maintained.

IPropertyStorage::SetTimes

For non-simple property sets only, sets the times supported by the underlying storage. The compound file storage implementation supports all three: modification, access, and creation. This implementation of **SetTimes** calls the **IStorage::SetElementTimes** method of the underlying storage to retrieve these times.

See Also

IPropertyStorage, **IStorage::SetElementTimes**

IProvideClassInfo Quick Info

The **IProvideClassInfo** interface provides a single method for accessing the type information for an object's **coclass** entry in its type library.

When to Implement

Implement this interface on any object that can provide type information for its entire class, that is, the **coclass** entry in the type library.

When to Use

Use this interface to access the **coclass** type information for an object.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IProvideClassInfo Methods	Description
GetClassInfo	Returns the **ITypeInfo** interface for the object's **coclass** type information.

IProvideClassInfo::GetClassInfo Quick Info

Returns a pointer to the **ITypeInfo** interface for the object's type information. The type information for an object corresponds to the object's **coclass** entry in a type library.

```
HRESULT GetClassInfo(
  ITypeInfo** ppTI      //Indirect pointer to object's type information
);
```

Parameters

ppTI

[out] Indirect pointer to object's type information. The caller is responsible for calling **ITypeInfo::Release** on the returned pointer if this method returns successfully.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The type information was successfully returned.

E_POINTER

The address in *ppTI* is not valid. For example, it may be NULL.

Remarks

Notes to Callers

The caller is responsible for calling **ITypeInfo::Release** when the returned interface pointer is no longer needed.

Notes to Implementers

This method must call **ITypeInfo::AddRef** before returning. If the object loads the type information from a type library, the type library itself will call **AddRef** in creating the pointer.

Because the caller cannot specify a locale identifier (LCID) when calling this method, this method must assume the neutral language, that is, LANGID_NEUTRAL, and use this value to determine what locale-specific type information to return.

This method must be implemented; E_NOTIMPL is not an acceptable return value.

IProvideClassInfo2 Quick Info

The **IProvideClassInfo2** interface is a simple extension to **IProvideClassInfo** for the purpose of making it quick and easy to retrieve an object's outgoing interface IID for its default event set. The mechanism, the added **GetGUID** method, is extensible for other types of GUIDs as well.

When to Implement

An object implements this interface to provide type information for its outgoing interfaces.

When to Use

Call the method in this interface to obtain type information on an object's outgoing interfaces.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IProvideClassInfo Method	Description
GetClassInfo	Returns the **ITypeInfo** interface for the object's **coclass** type information.

IProvideClassInfo2 Method	Description
GetGUID	Returns the GUID for the object's outgoing IID for its default event set.

IProvideClassInfo2::GetGUID Quick Info

Returns a GUID corresponding to the specified *dwGuidKind*. The *dwGuidKind* parameter has several values defined. See **GUIDKIND**. Additional flags can be defined at a later time and will be recognized by an **IProvideClassInfo2** implementation.

```
HRESULT GetGUID(
  DWORD dwGuidKind,     //Desired GUID
  GUID* pGUID           //Pointer to the desired GUID
);
```

Parameters

dwGuidKind

[in] Specifies the GUID desired on return. This parameter takes a value from the **GUIDKIND** enumeration.

pGUID

[out] Pointer to the caller's variable in which to store the GUID associated with *dwGuidKind*.

Return Values

S_OK

The GUID was successfully returned in **pGUID*.

E_POINTER

The address in *pGUID* is not valid (such as NULL).

E_UNEXPECTED

An unknown error occurred.

E_INVALIDARG

The *dwGuidKind* value does not correspond to a supported GUID kind.

Remarks

E_NOTIMPL is not a valid return code since it would be pointless to implement this interface without implementing this method.

See Also

GUIDKIND

IQuickActivate

Quick Info

The **IQuickActivate** interface allows controls and containers to avoid performance bottlenecks on loading controls. It combines the load-time or initialization-time handshaking between the control and its container into a single call.

When to Implement

Implement this interface on controls to achieve performance improvements during activation of the control.

If this interface is supported then all methods of this interface must be implemented.

When to Use

Containers call this interface on controls to achieve performance improvements in activating the control.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IQuickActivate Methods	Description
QuickActivate	Quick activates a control.
SetContentExtent	Sets the content extent of a control.
GetContentExtent	Gets the content extent of a control.

IQuickActivate::GetContentExtent Quick Info

Gets the content extent of the control.

```
HRESULT GetContentExtent(
  LPSIZEL psizel                //Pointer to size of the content extent
);
```

Parameters

psizel

[out] Pointer to size of the content extent.

Return Values

S_OK

The content extent was successfully returned.

IQuickActivate::QuickActivate Quick Info

Quick activates a control.

```
HRESULT QuickActivate(
    QACONTAINER*  pQaContainer,    //Pointer to container information structure
    QACONTROL*    pQaControl       //Pointer to control information structure
);
```

Parameters

pQaContainer

[in] Pointer to a structure containing information about the container.

pQaControl

[out] Pointer to a structure filled in by the control to return information about the control to the container. The container calling this method must reserve memory for this structure.

Return Values

S_OK

The quick activation is proceeding and the **QACONTROL** structure has been completed.

E_FAIL

An unexpected error occurred. The quick activation is not completed.

Remarks

If the control does not support **IQuickActivate**, the container performs certain handshaking operations when it loads the control. The container calls certain interfaces on the control and the control, in turn, calls back to certain interfaces on the container's client site. First, the container creates the control object and calls **QueryInterface** to query for interfaces that it needs. Then, the container calls **IOleObjectSetClientSite** on the control, passing a pointer to its client site. Next, the control calls **QueryInterface** on this site, retrieving a pointer to additional necessary interfaces.

Using the new **IQuickActivate::QuickActivate** method, the container passes a pointer to a **QACONTAINER** structure. The structure contains pointers to interfaces which are needed by the control and the values of some ambient properties that the control may need. Upon return, the control passes a pointer to a **QACONTROL** structure that contains pointers to its own interfaces that the container requires, and additional status information.

The **IPersist*::Load** and **IPersist*::InitNew** methods should be called after quick activation occurs. The control should establish its connections to the container's sinks during quick activation. However, these connections are not live until **IPersist*::Load** or **IPersist*::InitNew** has been called.

See Also

QACONTROL

IQuickActivate::SetContentExtent

Quick Info

Sets the content extent of the control.

```
HRESULT SetContentExtent(
  LPSIZEL psizel                //Size of the content extent
);
```

Parameter

psizel

[in] Size of the content extent.

Return Values

S_OK

The extent was successfully set.

E_FAIL

The object's size is fixed and the extent cannot be set.

IRootStorage Quick Info

The **IRootStorage** interface contains a single method that switches a storage object to a different underlying file and saves the storage object to that file. The save operation occurs even with low memory conditions and uncommitted changes to the storage object. A subsequent call to **IStorage::Commit** is guaranteed to not consume any additional memory.

When to Implement

Storage objects that are based on a file should implement **IRootStorage** in addition to the **IStorage** interface. For storage objects that are not file-based, this interface is not necessary.

OLE provides an implementation of a storage object, including the **IRootStorage** interface, as part of its compound file implementation.

When to Use

The primary use for the **IRootStorage** interface is to save a storage object to a file during low memory conditions. Typically, the container application calls the **IRootStorage** interface to switch to a new file.

If you have an **IStorage** pointer to a compound file object, you can call **IStorage::QueryInterface** with *IID_IRootStorage* to obtain a pointer to the **IRootStorage** interface.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IRootStorage Method	Description
SwitchToFile	Copy the file underlying this root storage object, then associate this storage with the copied file.

See Also

IStorage, StgCreateDocfile

IRootStorage::SwitchToFile Quick Info

Copies the current file associated with the storage object to a new file. The new file is then used for the storage object and any uncommitted changes.

```
HRESULT SwitchToFile(
  LPOLESTR pszFile          //Filename for the new file
);
```

Parameter

pszFile

Specifies the filename for the new file. It cannot be the name of an existing file. If NULL, this method creates a temporary file with a unique name, and you can call **IStorage::Stat** to retrieve the name of the temporary file.

Return Values

S_OK

The file was successfully copied.

STG_E_MEDIUMFULL

The file was not copied because of insufficient space on the storage device.

STG_E_ACCESSDENIED

The file was not copied because the caller does not have permission to access storage device.

STG_E_INVALIDPOINTER

The file was not copied because the *pszFile* pointer is invalid.

STG_E_FILEALREADYEXISTS

The file was not copied because the new filename (*pszFile*) points to an existing file.

Remarks

The **IRootStorage::SwitchToFile** method copies the file associated with the storage object. An OLE container calls **SwitchToFile** to perform a full save on a file in a low-memory situation. Typically, this is done only after a normal full save operation (i.e., save to temporary file, delete original file, rename temporary file) has failed with an E_OUTOFMEMORY error.

It is illegal to call **SwitchToFile** if the storage object or anything contained within it has been marshalled to another process. As a consequence, before calling **SwitchToFile**, the container must call the **IPersistStorage::HandsOffStorage** method for any element within the storage object that is loaded or running. The **HandsOffStorage** method forces the element to release its storage pointers and enter the hands-off storage mode. The container must also release all pointers to streams or storages that are contained in this root storage. After the full save operation is completed, the container returns the contained elements to normal storage mode.

Notes to Implementers

If you are implementing your own storage objects, the **IRootStorage** methods (including **QueryInterface**, **AddRef**, and **Release**) must not consume additional memory or file handles.

See Also

IPersistStorage::HandsOffStorage, **IPersistStorage::SaveCompleted**, **IStorage::Commit**, **IStorage::Stat**

IRootStorage - Compound File Implementation

OLE's compound file implementation of **IRootStorage** provides a way to support saving files in low-memory or low disk-space situations. For information on how this interface behaves, see **IRootStorage**.

When to Use

Use the system-supplied implementation of **IRootStorage** only to support saving files under low memory conditions.

Remarks

It is possible to call OLE's implementation of **IRootStorage::SwitchToFile** to do a normal Save As operation to

another file. Applications that do so, however, may not be compatible with future generations of OLE storage. To avoid this possibility, applications performaing a Save As operation should manually create the second docfile and invoke

IStorage::CopyTo. IRootStorage::SwitchToFile should beused only in emergency (low memory or disk space) situations.

See Also

IRootStorage; **IRootStorage::SwitchToFile**

IROTData

Quick Info

The **IROTData** interface is implemented by monikers to enable the Running Object Table (ROT) to compare monikers against each other.

The ROT uses the **IROTData** interface to test whether two monikers are equal. The ROT must do this when, for example, it checks whether a specified moniker is registered as running.

When to Implement

You must implement **IROTData** if you are writing your own moniker class (that is, writing your own implementation of the **IMoniker** interface), and if your monikers are meant to be registered in the ROT.

When to Use

You typically do not need to use this interface. This interface is used by the system's implementation of the ROT.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IROTData Method	Description
GetComparisonData	Retrieve data to allow moniker to be compared with another.

See Also

IMoniker, IRunningObjectTable

IROTData::GetComparisonData Quick Info

Retrieves data from a moniker that can be used to test the moniker for equality against another moniker.

```
HRESULT GetComparisonData(
  PVOID *ppvData,     //Indirect pointer to a buffer that receives the comparison
                      data
  ULONG cbMax,        //Length of buffer
  PULONG pcbData      //Pointer to the length of the comparison data
);
```

Parameters

ppvData

[out] Indirect pointer to a buffer that receives the comparison data.

cbMax

[in] Length of the buffer specified in *ppvData*.

pcbData

[out] Pointer to the length of the comparison data.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The comparison data was successfully returned.

Remarks

The **IROTData::GetComparisonData** method is primarily called by the Running Object Table (ROT). The comparison data returned by the method is tested for binary equality against the comparison data returned by another moniker. The *pcbData* parameter enables the ROT to locate the end of the data returned.

Notes to Implementers

The comparison data that you return must uniquely identify the moniker, while still being as short as possible. The comparison data should include information about the internal state of the moniker, as well as the moniker's CLSID. For example, the comparison data for a file moniker would include the path name stored within the moniker, as well as the CLSID of the file moniker implementation. This makes it possible to distinguish two monikers that happen to store similar state information but are instances of different moniker classes.

The comparison data for a moniker cannot exceed 2048 bytes in length. For composite monikers, the total length of the comparison data for all of its components cannot exceed 2048 bytes; consequently, if your moniker can be a component within a composite moniker, the comparison data you return must be significantly less than 2048 bytes.

If your comparison data is longer than the value specified by the *cbMax* parameter, you must return an error. Note that when **IROTData::GetComparisionData** is called on the components of a composite

moniker, the value of *cbMax* becomes smaller for each moniker in sequence.

See Also

IMoniker, **IRunningObjectTable**

IRunnableObject Quick Info

The **IRunnableObject** interface enables a container to control the running of its embedded objects. In the case of an object implemented with a local server, calling **IRunnableObject::Run** launches the server's .EXE file. In the case of an object implemented with an in-process server, calling the **Run** method causes the object .DLL file to transition into the running state.

When to Implement

Object handlers should implement **IRunnableObject** to provide their containers with a way to run them and manage their running state. DLL object applications should implement **IRunnableObject** to support silent updates of their objects.

When to Use

Containers call **IRunnableObject** to determine if an embedded object is running, to force an object to run, to lock an object into the running state, or to inform an object handler whether its object is being run as either a simple embedding or as a link source.

Methods VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IRunnableObject Methods	Description
GetRunningClass	Returns CLSID of a running object.
Run	Forces an object to run.
IsRunning	Determines if an object is running.
LockRunning	Locks an object into running state.
SetContainedObject	Indicates that an object is embedded.

IRunnableObject::GetRunningClass Quick Info

Returns the CLSID of a running object.

```
HRESULT GetRunningClass(
  LPCLSID lpClsid      //Pointer to an object's CLSID
);
```

Parameter

lpClsid

[out] Pointer to the object's class identifier.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

CLSID was returned successfully.

Remarks

If an embedded document was created by an application that is not available on the user's computer, the document, by a call to **CoTreatAsClass**, may be able to display itself for editing by emulating a class that *is* supported on the user's machine. In this case, the CLSID returned by a call to **IRunnableObject::GetRunningClass** will be that of the class being emulated, rather than the document's native class.

See Also

CoTreatAsClass

IRunnableObject::IsRunning Quick Info

Determines whether an object is currently in the running state.

BOOL IsRunning();

Return Values

TRUE

The object is in the running state.

FALSE

The object is not in the running state.

Remarks

A container application could call **IRunnableObject::IsRunning** when it needs to know if the server is immediately available. For example, a container's implementation of the **IOleItemContainer::GetObject** method would return an error if the server is not running and the bindspeed parameter specifies BINDSPEED_IMMEDIATE.

An object handler could call **IRunnableObject::IsRunning** when it wants to avoid conflicts with a running server or when the running server might have more up-to-date information. For example, a handler's implementation of **IOleObject::GetExtent** would delegate to the object server if it is running, because the server's information might be more current than that in the handler's cache.

OleIsRunning is a helper function that conveniently repackages the functionality offered by **IRunnableObject::IsRunning**. With the release of OLE 2.01, the implementation of **OleIsRunning** was changed so that it calls **QueryInterface**, asks for **IRunnableObject**, and then calls **IRunnableObject::IsRunning**. In other words, you can use the interface and the helper function interchangeably.

See Also

OleIsRunning

IRunnableObject::LockRunning Quick Info

Locks an already running object into its running state or unlocks it from its running state.

```
HRESULT LockRunning(
    BOOL fLock,                 //Flag indicating whether object is locked
    BOOL fLastUnlockCloses      //Flag indicating whether to close object
);
```

Parameters

fLock

[in] TRUE locks the object into its running state. FALSE unlocks the object from its running state.

fLastUnlockCloses

[in] TRUE specifies that if the connection being released is the last external lock on the object, the object should close. FALSE specifies that the object should remain open until closed by the user or another process.

Return Values

This method supports the standard return values E_FAIL, E_INVALIDARG, E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

If the value of *fLock* is TRUE, the object was successfully locked; if the value of *fLock* is FALSE, the object was successfully unlocked.

Remarks

Most implementations of **IRunnableObject::LockRunning** call **CoLockObjectExternal**.

OleLockRunning is a helper function that conveniently repackages the functionality offered by **IRunnableObject::LockRunning**. With the release of OLE 2.01, the implementation of **OleLockRunning** was changed to call **QueryInterface**, ask for **IRunnableObject**, and then call **IRunnableObject::LockRunning**. In other words, you can use the interface and the helper function interchangeably.

See Also

CoLockObjectExternal

IRunnableObject::Run Quick Info

Runs an object.

```
HRESULT Run(
  LPBC lpbc      //Pointer to binding context
);
```

Parameter

lpbc

[in] Pointer to the binding context of the run operation. May be NULL.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The object was successfully placed in the running state.

Remarks

Containers call **IRunnableObject::Run** to force their objects to enter the running state. If the object is not already running, calling **IRunnableObject::Run** can be an expensive operation, on the order of many seconds. If the object is already running, then this method has no effect on the object.

Notes to Callers

When called on a linked object that has been converted to a new class since the link was last activated, **IRunnableObject::Run** may return OLE_E_CLASSDIFF. In this case, the client should call **IOleLink::BindToSource**.

OleRun is a helper function that conveniently repackages the functionality offered by **IRunnableObject::Run**. With the release of OLE 2.01, the implementation of **OleRun** was changed so that it calls **QueryInterface**, asks for **IRunnableObject**, and then calls **IRunnableObject::Run**. In other words, you can use the interface and the helper function interchangeably.

Notes to Implementers

The object should register in the running object table if it has a moniker assigned. The object should not hold any strong locks on itself; instead, it should remain in the unstable, unlocked state. The object should be locked when the first external connection is made to the object.

An embedded object must hold a lock on its embedding container while it is in the running state. The Default handler provided by OLE 2 takes care of locking the embedding container on behalf of objects implemented by an EXE object application. Objects implemented by a DLL object application must explicitly put a lock on their embedding containers, which they do by first calling **IOleClientSite::Getcontainer** to get a pointer to the container, then calling **IOleContainer::LockContainer** to actually place the lock. This lock must be released when **IOleObject::Close** is called.

See Also

IOleLink::BindToSource, **OleRun**

IRunnableObject::SetContainedObject Quick Info

Notifies an object that it is embedded in an OLE container, which ensures that reference counting is done correctly for containers that support links to embedded objects.

```
HRESULT SetContainedObject(
  BOOL fContained      //Flag indicating whether object is embedded
);
```

Parameter

fContained

[in] TRUE specifies that the object is contained in an OLE container. FALSE indicates that it is not.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY AND E_UNEXPECTED, as well as the following:

S_OK

Object has been marked as a contained embedding.

Remarks

The **IRunnableObject::SetContainedObject** method enables a container to inform an object handler that it is embedded in the container, rather than acting as a link. This call changes the container's reference on the object from strong, the default for external connections, to weak. When the object is running visibly, this method is of little significance because the end user has a lock on the object. During a silent update of an embedded link source, however, the container should not be able to hold an object in the running state after the link has been broken. For this reason, the container's reference to the object must be weak.

Notes to Callers

A container application must call **IRunnableObject::SetContainedObject** if it supports linking to embedded objects. It normally makes the call immediately after calling **OleLoad** or **OleCreate** and never calls the method again, even before it closes. Moreover, a container almost always calls this method with *fContained* set to TRUE. The use of this method with *fContained* set to FALSE is rare.

Calling **IRunnableObject::SetContainedObject** is optional only when you know that the embedded object will not be referenced by any client other than the container. If your container application does not support linking to embedded objects; it is preferable, but not necessary, to call **IRunnableObject::SetContainedObject**.

OleSetContainedObject is a helper function that conveniently repackages the functionality offered by **IRunnableObject::SetContainedObject**. With the release of OLE 2.01, the implementation of **OleSetContainedObject** was changed to call **QueryInterface**, ask for **IRunnableObject**, and then call **IRunnableObject::SetContainedObject**. In other words, you can use the interface and the helper function interchangeably.

See Also

OleSetContainedObject, **OleNoteObjectVisible**, **CoLockObjectExternal**

IRunningObjectTable Quick Info

The **IRunningObjectTable** interface manages access to the Running Object Table (ROT), a globally accessible look-up table on each workstation. A workstation's ROT keeps track of those objects that can be identified by a moniker and that are currently running on the workstation. When a client tries to bind a moniker to an object, the moniker checks the ROT to see if the object is already running; this allows the moniker to bind to the current instance instead of loading a new one.

The ROT contains entries of the form:

```
(pmkObjectName, pUnkObject)
```

The *pmkObjectName* element is a pointer to the moniker that identifies the running object. The *pUnkObject* element is a pointer to the running object itself. During the binding process, monikers consult the *pmkObjectName* entries in the Running Object Table to see if an object is already running.

Objects that can be named by monikers must be registered with the ROT when they are loaded and their registration must be revoked when they are no longer running.

When to Implement

You do not need to implement this interface. The system provides an implementation of the Running Object Table that is suitable for all situations.

When to Use

You typically use the ROT if you're a moniker provider (that is, you hand out monikers identifying your objects to make them accessible to others) or if you're writing your own moniker class (that is, implementing the **IMoniker** interface).

If you are a moniker provider, you register your objects with the ROT when they begin running and revoke their registrations when they are no longer running. This enables the monikers that you hand out to be bound to running objects. You should also use the ROT to record the object's last modification time. You can get an **IRunningObjectTable** interface pointer to the local ROT by calling the **GetRunningObjectTable** function.

The most common type of moniker provider is a compound-document link source. This includes server applications that support linking to their documents (or portions of a document) and container applications that support linking to embeddings within their documents. Server applications that do not support linking can also use the ROT to cooperate with container applications that support linking to embeddings.

If you are writing your own moniker class, you use the ROT to determine whether a object is running and to retrieve the object's last modification time. You can get an **IRunningObjectTable** interface pointer to the local ROT by calling the **IBindCtx::GetRunningObjectTable** method on the bind context for the current binding operation. Moniker implementations should always use the bind context to acquire a pointer to the ROT; this allows future implementations of **IBindCtx** to modify binding behavior. Note that you must also implement the **IROTData** interface on your moniker class in order to allow your monikers to be registered with the ROT.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IRunningObjectTable Methods	Description
Register	Registers an object with the ROT.
Revoke	Revokes an object's registration with the ROT.
IsRunning	Checks whether an object is running.
GetObject	Returns a pointer to an object given its moniker.
NoteChangeTime	Notifies the ROT that an object has changed.
GetTimeOfLastChange	Returns the time an object was last changed.
EnumRunning	Returns an enumerator for the ROT.

See Also

IBindCtx::GetRunningObjectTable, **IROTData**, **GetRunningObjectTable**

IRunningObjectTable::EnumRunning Quick Info

Creates and returns a pointer to an enumerator that can list the monikers of all the objects currently registered in the Running Object Table (ROT).

```
HRESULT EnumRunning(

  IEnumMoniker **ppenumMoniker      //Indirect pointer to the enumerator for ROT
);
```

Parameter

ppenumMoniker

[out] When successful, indirect pointer to the **IEnumMoniker** interface on the new enumerator. In this case, the implementation calls **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If an error occurs; the implementation sets *ppenumMoniker* to NULL.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

An enumerator was successfully returned.

Remarks

IRunningObjectTable::EnumRunning must create and return a pointer to an **IEnumMoniker** interface on an enumerator object. The standard enumerator methods can then be called to enumerate the monikers currently registered in the registry. The enumerator cannot be used to enumerate monikers that are registered in the ROT after the enumerator has been created.

The **EnumRunning** method is intended primarily for the use by the system in implementing the Alert Object Table. Note that OLE 2 does not include an implementation of the Alert Object Table.

See Also

IEnum*XXXX*, **IEnumMoniker**

IRunningObjectTable::GetObject Quick Info

Determines whether the object identified by the specified moniker is running, and if it is, retrieves a pointer to that object. This method looks for the moniker in the Running Object Table (ROT), and retrieves the pointer registered there.

```
HRESULT GetObject(
  IMoniker *pmkObjectName,     //Pointer to the moniker on the object
  IUnknown **ppunkObject       //Indirect pointer to the object
);
```

Parameters

pmkObjectName

[in] Pointer to the moniker to search for in the Running Object Table.

ppunkObject

[out] When successful. indirect pointer to the **IUnknown** interface on the running object. In this case, the implementation calls **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release**. If the object is not running or if an error occurs, the implementation sets *ppunkObject* to NULL.

Return Values

S_OK

Indicates that *pmkObjectName* was found in the ROT and a pointer was returned.

S_FALSE

There is no entry for *pmkObjectName* in the ROT, or that the object it identifies is no longer running (in which case, the entry is revoked).

Remarks

This method checks the ROT for the moniker specified by *pmkObjectName*. If that moniker had previously been registered with a call to **IRunningObjectTable::Register**, this method returns the pointer that was registered at that time.

Notes to Callers

Generally, you call the **IRunningObjectTable::GetObject** method only if you are writing your own moniker class (that is, implementing the **IMoniker** interface). You typically call this method from your implementation of **IMoniker::BindToObject**.

However, note that not all implementations of **IMoniker::BindToObject** need to call this method. If you expect your moniker to have a prefix (indicated by a non-NULL *pmkToLeft* parameter to **IMoniker::BindToObject**), you should not check the ROT. The reason for this is that only complete monikers are registered with the ROT, and if your moniker has a prefix, your moniker is part of a composite and thus not complete. Instead, your moniker should request services from the object identified by the prefix (for example, the container of the object identified by your moniker).

See Also

IMoniker::BindToObject

IRunningObjectTable::GetTimeOfLastChange Quick Info

Returns the time that an object was last modified. The object must have previously been registered with the Running Object Table (ROT). This method looks for the last change time recorded in the ROT.

```
HRESULT GetTimeOfLastChange(
  IMoniker *pmkObjectName,     //Pointer to moniker on the object whose status is desired
  FILETIME *pfiletime          //Pointer to structure that receives object's last change time
);
```

Parameters

pmkObjectName

[in] Pointer to the **IMoniker** interface on the moniker to search for in the ROT.

pfiletime

[out] Pointer to a **FILETIME** structure that receives the object's last change time.

Return Values

S_OK

The last change time was successfully retrieved.

S_FALSE

There is no entry for *pmkObjectName* in the ROT, or that the object it identifies is no longer running (in which case, the entry is revoked).

Remarks

This method returns the change time that was last reported for this object by a call to **IRunningObjectTable::NoteChangeTime**. If **IRunningObjectTable::NoteChangeTime** has not been called previously, the method returns the time that was recorded when the object was registered.

This method is provided to enable checking whether a connection between two objects (represented by one object holding a moniker that identifies the other) is up-to-date. For example, if one object is holding cached information about the other object, this method can be used to check whether the object has been modified since the cache was last updated. See **IMoniker::GetTimeOfLastChange**.

Notes to Callers

Generally, you call **IRunningObjectTable::GetTimeOfLastChange** only if you are writing your own moniker class (that is, implementing the **IMoniker** interface). You typically call this method from your implementation of **IMoniker::GetTimeOfLastChange**. However, you should do so only if the *pmkToLeft* parameter of **IMoniker::GetTimeOfLastChange** is NULL. Otherwise, you should call **IMoniker::GetTimeOfLastChange** on your *pmkToLeft* parameter instead.

See Also

IMoniker::GetTimeOfLastChange, **IRunningObjectTable::NoteChangeTime**

IRunningObjectTable::IsRunning Quick Info

Determines whether the object identified by the specified moniker is currently running. This method looks for the moniker in the Running Object Table (ROT).

```
HRESULT IsRunning(
  IMoniker *pmkObjectName      //Pointer to the moniker of the object whose status is desired
);
```

Parameter

pmkObjectName

[in] Pointer to the **IMoniker** interface on the moniker to search for in the Running Object Table.

Return Values

S_OK

The object identified by *pmkObjectName* is running.

S_FALSE

There is no entry for *pmkObjectName* in the ROT, or that the object it identifies is no longer running (in which case, the entry is revoked).

Remarks

This method simply indicates whether a object is running. To retrieve a pointer to a running object, use the **IRunningObjectTable::GetObject** method.

Notes to Callers

Generally, you call the **IRunningObjectTable::IsRunning** method only if you are writing your own moniker class (that is, implementing the **IMoniker** interface). You typically call this method from your implementation of **IMoniker::IsRunning**. However, you should do so only if the *pmkToLeft* parameter of **IMoniker::IsRunning** is NULL. Otherwise, you should call **IMoniker::IsRunning** on your *pmkToLeft* parameter instead.

See Also

IMoniker::IsRunning

IRunningObjectTable::NoteChangeTime Quick Info

Records the time that a running object was last modified. The object must have previously been registered with the Running Object Table (ROT). This method stores the time of last change in the ROT.

```
HRESULT NoteChangeTime(
  DWORD dwRegister,      //Value identifying registration being updated
  FILETIME *pfiletime    //Pointer to structure containing object's last change time
);
```

Parameters

dwRegister

[in] Value identifying the ROT entry of the changed object. This value was previously returned by **IRunningObjectTable::Register**.

pfiletime

[in] Pointer to a **FILETIME** structure containing the object's last change time.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The change time was recorded successfully.

Remarks

The time recorded by this method can be retrieved by calling **IRunningObjectTable::GetTimeOfLastChange**.

This method is provided to enable a program to check whether a connection between two objects (represented by one object holding a moniker that identifies the other) is up-to-date. For example, if one object is holding cached information about the other object, this method can be used to check whether the object has been modified since the cache was last updated. See **IMoniker::GetTimeOfLastChange**.

Notes to Callers

If you're a moniker provider (that is, you hand out monikers identifying your objects to make them accessible to others), you must call the **IRunningObjectTable::NoteChangeTime** method whenever your objects are modified. You must have previously called **IRunningObjectTable::Register** and stored the identifier returned by that method; you use that identifier when calling **IRunningObjectTable::NoteChangeTime**.

The most common type of moniker provider is a compound-document link source. This includes server applications that support linking to their documents (or portions of a document) and container applications that support linking to embeddings within their documents. Server applications that do not support linking can also use the ROT to cooperate with container applications that support linking to embeddings.

When an object is first registered in the ROT, the ROT records its last change time as the value returned by calling **IMoniker::GetTimeOfLastChange** on the moniker being registered.

See Also

IRunningObjectTable::GetTimeOfLastChange, IMoniker::GetTimeOfLastChange

IRunningObjectTable::Register Quick Info

Registers an object and its identifying moniker in the Running Object Table (ROT).

```
HRESULT Register(
  DWORD grfFlags,              //Specifies a weak or a strong reference
  IUnknown *punkObject,        //Pointer to the object being registered
  IMoniker *pmkObjectName,     //Pointer to the moniker of the object being registered
  DWORD *pdwRegister           //Pointer to the value identifying the registration
);
```

Parameters

grfFlags

[in] Specifies whether the ROT's reference to *punkObject* is weak or strong. This value must be either zero, indicating a weak reference that does not call **IUnknown::AddRef**; or ROTFLAGS_REGISTRATIONKEEPSALIVE, indicating a strong reference that calls **IUnknown::AddRef** and can keep the object running. If a strong reference is registered, a strong reference is released when the object's registration is revoked. Most callers specify zero, indicating a weak reference.

punkObject

[in] Pointer to the object that is being registered as running.

pmkObjectName

[in] Pointer to the moniker that identifies *punkObject*.

pdwRegister

[out] Pointer to a 32-bit value that can be used to identify this ROT entry in subsequent calls to **IRunningObjectTable::Revoke** or **IRunningObjectTable::NoteChangeTime**. The caller cannot specify NULL for this parameter. If an error occurs, **pdwRegister* is set to zero.

Return Values

This method supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The object was successfully registered.

MK_S_MONIKERALREADYREGISTERED

The moniker/object pair was successfully registered, but that another object (possibly the same object) has already been registered with the same moniker.

Remarks

This method registers a pointer to an object under a moniker that identifies the object. The moniker is used as the key when the table is searched with **IRunningObjectTable::GetObject**.

Registering a second object with the same moniker, or re-registering the same object with the same moniker, creates a second entry in the ROT. In this case, **IRunningObjectTable::Register** returns

MK_S_MONIKERALREADYREGISTERED. Each call to **IRunningObjectTable::Register** must be matched by a call to **IRunningObjectTable::Revoke** because even duplicate entries have different *pdwRegister* identifiers. A problem with duplicate registrations is that there is no way to determine which object will be returned if the moniker is specified in a subsequent call to **IRunningObjectTable::IsRunning**.

Notes to Callers

If you're a moniker provider (that is, you hand out monikers identifying your objects to make them accessible to others), you must call the **IRunningObjectTable::Register** method to register your objects when they begin running. You must also call this method if you rename your objects while they are loaded.

The most common type of moniker provider is a compound-document link source. This includes server applications that support linking to their documents (or portions of a document) and container applications that support linking to embeddings within their documents. Server applications that do not support linking can also use the ROT to cooperate with container applications that support linking to embeddings.

If you're writing a server application, you should register an object with the ROT when it begins running, typically in your implementation of **IOleObject::DoVerb**. The object must be registered under its full moniker, which requires getting the moniker of its container document using **IOleClientSite::GetMoniker**. You should also revoke and re-register the object in your implementation of **IOleObject::SetMoniker**, which is called if the container document is renamed.

If you're writing a container application that supports linking to embeddings, you should register your document with the ROT when it is loaded. If your document is renamed, you should revoke and re-register it with the ROT and call **IOleObject::SetMoniker** for any embedded objects in the document to give them an opportunity to re-register themselves.

You must cache the identifier returned in *pdwRegister*, and use it in a call to **IRunningObjectTable::Revoke** to revoke the registration when the object is no longer running or when its moniker changes. This revocation is important because there is no way for the system to automatically remove entries from the ROT.

The system's implementation of **IRunningObjectTable::Register** calls **IMoniker::Reduce** on the *pmkObjectName* parameter to ensure that the moniker is fully reduced before registration. If a object is known by more than one fully reduced moniker, then it should be registered under all such monikers.

See Also

IMoniker::Reduce, **IOleClientSite::GetMoniker**, **IOleObject::SetMoniker**, **IRunningObjectTable::IsRunning**, **IRunningObjectTable::Revoke**

IRunningObjectTable::Revoke Quick Info

Removes from the Running Object Table (ROT) an entry that was previously registered by a call to **IRunningObjectTable::Register**.

```
HRESULT Revoke(
  DWORD dwRegister      //Value identifying registration to be revoked
);
```

Parameter

dwRegister

[in] Value identifying the ROT entry to revoke. This value was previously returned by **IRunningObjectTable::Register**.

Return Values

This method supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The object's registration was successfully revoked.

Remarks

This method undoes the effect of a call to **IRunningObjectTable::Register**, removing both the moniker and the pointer to the object identified by that moniker.

Notes to Callers

If you're a moniker provider (that is, you hand out monikers identifying your objects to make them accessible to others), you must call the **IRunningObjectTable::Revoke** method to revoke the registration of your objects when they stop running. You must have previously called **IRunningObjectTable::Register** and stored the identifier returned by that method; you use that identifier when calling **IRunningObjectTable::Revoke**.

The most common type of moniker provider is a compound-document link source. This includes server applications that support linking to their documents (or portions of a document) and container applications that support linking to embeddings within their documents. Server applications that do not support linking can also use the ROT to cooperate with container applications that support linking to embeddings.

If you're writing a container application, you must revoke a document's registration when the document is closed. You must also revoke a document's registration before re-registering it when it is renamed.

If you're writing a server application, you must revoke an object's registration when the object is closed. You must also revoke an object's registration before re-registering it when its container document is renamed (see **IOleObject::SetMoniker**).

See Also

IOleObject::SetMoniker, **IRunningObjectTable::Register**

IServerSecurity Quick Info

Used by a server to help identify the client and to manage impersonation of the client. **IServerSecurity:QueryBlanket** and **IServerSecurity::ImpersonateClient** may only be called before the ORPC call completes. The interface pointer must be released when it is no longer needed.

When a client calls a server, the server can call **CoGetCallContext** until the server sends the reply back to the client. The pointer to the instance of **IServerSecurity** returned by **CoGetCallContext** is automaticly deleted when the server sends the reply back to the client.

When to Implement

The stub managment code in the system provides an implementation of **IServerSecurity** for objects by default as part of each incoming call, so typically you would not implement this interface.

You may choose to implement **IServerSecurity** on the custom stubs of objects that support custom marshaling to maintain a consistent programming model for their objects.

When to Use

The methods of the **IServerSecurity** interface are called by the server/object to examine or alter the security level of the connection between the caller and this particular object. Its most common use is for impersonation (**IServerSecurity::ImpersonateClient** and **::RevertToSelf**), where the server impersonates the client to test the privilege level of the calling client with an **AccessCheck** call. The information obtained through **IServerSecurity** also allows an object to implement its own security framework, perhaps not based on the Access Control Lists (ACLs) that impersonation is geared toward. A different implementation could base its security framework on the client name or other information available through a call to the **QueryBlanket** method.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IServerSecurity Methods	Description
QueryBlanket	Called by the server to find out about the client that invoked one of its methods.
ImpersonateClient	Allows a server to impersonate a client for the duration of a call.
RevertToSelf	Restores the authentication information on a thread to the process's identity.
IsImpersonating	Indicates whether the server is currently impersonating the client.

See Also

Security in COM

IServerSecurity::ImpersonateClient Quick Info

Allows a server to impersonate a client for the duration of a call.

HRESULT ImpersonateClient()

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

Success.

Remarks

IServerSecurity::ImpersonateClient allows a server to impersonate a client for the duration of a call. What the server may do depends on the impersonation level, specified through one of the **RPC_C_IMP_LEVEL_*xxx*** constants. The server may impersonate the client on any secure call at identify, impersonate, or delegate level. At identify level, the server may only find out the client name and perform ACL checks; it may not access system objects as the client. At delegate level, the server may make off-machine calls while impersonating the client. The impersonation information only lasts until the end of the current method call. At that time, **IServerSecurity::RevertToSelf** will automatically be called if necessary.

Traditionally, impersonation information is not nested - the last call to any Win32 impersonation mechanism overrides any previous impersonation. However, in the apartment model, impersonation is maintained during nested calls. Thus if the server ***A*** receives a call from ***B***, impersonates, calls ***C***, receives a call from ***D***, impersonates, reverts, and receives the reply from ***C***, the impersonation will be set back to ***B***, not ***A***.

Distributed COM currently does not support dynamic impersonation. The only way to change the client token associated with remote OLE calls is to use **IClientSecurity::SetBlanket** on the proxy being called. Calling **IServerSecurity::ImpersonateClient** to impersonate your client and then making a remote call to another server will not affect the token the second server sees when it impersonates on your call.

See Also

CoImpersonateClient

IServerSecurity::IsImpersonating Quick Info

Indicates whether **IServerSecurity::ImpersonateClient** has been called without a matching call to **IServerSecurity::RevertToSelf**.

BOOL IsImpersonating()

Return Values

TRUE

This thread has called **IServerSecurity::ImpersonateClient** and is currently impersonating the client of this call.

FALSE

This thread is not currently impersonating the client of this call.

See Also

IServerSecurity::RevertToSelf.

IServerSecurity::QueryBlanket Quick Info

Called by the server to find out about the client that invoked one of its methods.

```
HRESULT QueryBlanket(
    DWORD*    pAuthnSvc,          //Pointer to the current authentication service
    DWORD*    pAuthzSvc,          //Pointer to the current authorization service
    OLECHAR **    pServerPrincNam,    //Pointer to the current principal name
    DWORD * pAuthnLevel,          //Pointer to the current authentication level
    DWORD * pImpLevel,            //Must be NULL
    RPC_AUTHZ_HANDLE *  pPrivs,   //Pointer to string identifying client
    DWORD **    pCapabilities       //Pointer to flags indicating further capabilities of the proxy
);
```

Parameter

pAuthnSvc

[out] Pointer to the current *authentication* service. This will be a single value taken from the list of **RPC_C_AUTHN_xxx** constants. May be NULL, in which case the current authentication service is not returned.

pAuthzSvc

[out] Pointer to the current *authorization* service. This will be a single value taken from the list of **RPC_C_AUTHZ_*xxx*** constants. May be NULL, in which case the current authorization service is not returned.

pServerPrincName

[out] Pointer to the current principal name. The string will be allocated by the callee using **CoTaskMemAlloc**, and must be freed by the caller using **CoTaskMemFree** when they are done with it. May be NULL, in which case the principal name is not returned.

pAuthnLevel

[out] Pointer to the current authentication level. This will be a single value taken from the list of **RPC_C_AUTHN_LEVEL_*xxx*** constants. May be NULL, in which case the current authorization level is not returned.

pImpLevel

[out] Must be NULL; the current authentication level is not supplied.

pPrivs

[out] Pointer to a string identifying the client. For NTLMSSP the string is of the form *domain\user*. This is not a copy, and so should not be modified or freed, and is not valid after the ORPC call completes.

pCapabilities

[out] Pointer to return flags indicating further capabilities of the call. Currently, no flags are defined for this parameter and it will only return EOAC_NONE. May be NULL, in which case the flags indicating further capabilities are not returned.

Return Values

This method supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

Success.

Remarks

IServerSecurity::QueryBlanket is used by the server to find out about the client that invoked one of its methods. To get a pointer to **IServerSecurity** for the current call on the current thread, call **CoGetCallContext**, specifying **IID_IServerSecurity**. This interface pointer may only be used in the same apartment as the call for the duration of the call.

See Also

CoGetCallContext

IServerSecurity::RevertToSelf Quick Info

Restores the authentication information on a thread to the process's identity.

HRESULT RevertToSelf()

Return Values

This method supports the standard return value E_FAIL, as well as the following:

S_OK

Success.

Remarks

IServerSecurity::RevertToSelf restores the authentication information and reverts an impersonation on a thread to the process's identity. This method will only revert impersonation changes made by **IServerSecurity::ImpersonateClient**. If the thread token is modified by other means (through the **SetThreadToken** or **RpcImpersonateClient** Win32 functions) the result of this function is undefined.

In the apartment model, **CoRevertToSelf** (**IServerSecurity::RevertToSelf**) affects only the current method invocation. If there are nested method invocations, they each may have their own impersonation and COM will correctly restore the impersonation before returning to them (regardless of whether or not **CoRevertToSelf**/**IServerSecurity::RevertToSelf** was called).

See Also

CoRevertToSelf

IServiceProvider

[New - Windows NT]

The **IServiceProvider** interface locates a service specified by its GUID and returns the interface pointer for the requested interface on the service.

When to Implement

An object that provides services should implement the **IServiceProvider** interface as a general way to supply its clients with interface pointers to the interfaces on the service.

A service is often provided through a separate object from the client site. For example, the service can be provided through a separate control or some other object that the client can communicate with.

Usually, the client communicates through its client site object in the container. The container calls **IOleObject::SetClientSite** to provide a pointer to the **IOleClientSite** interface for the embedded object's client site. Then, the client must call methods in the **IOleClientSite** interface to find out about services that its container supports. Thus, the client site must provide a way for the client to access the service when necessary, even if the service is provided through a separate object.

For example, an in-place object calls **IOleInPlaceSite::GetWindowContext** to obtain interface pointers for the document object that contains the site and for the frame object that contains the document. Both of these interface pointers are on objects separate from the site object, so the client cannot call **IOleInPlaceSite::QueryInterface** to obtain these interface pointers.

The **IServiceProvider** interface is a general way to provide interface pointers for services, so that the site object need not implement ad hoc solutions as the need arises.

When to Use

This interface itself has only one method, **IServiceProvider::QueryService**. The caller specifies a GUID for the service and the IID of the requested interface. The interface pointer is returned in a caller-supplied variable.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IServiceProvider Methods	Description
QueryService	Returns an interface pointer to the requested interface on a service.

See Also

IServiceProvider::QueryService

Creates or accesses the specified service and returns an interface pointer to the specified interface for the service.

```
HRESULT QueryService(
  REFGUID guidService,     //Unique identifier for the requested service
  REFIID riid,             //Unique identifier for the requested interface
  void** ppv               //Indirect pointer to the interface on the service
);
```

Parameter

guidService

[in] Unique identifier for the requestedservice.

riid

[in] Unique identifier for the requested interface on the service.

ppv

[out] Indirect pointer to the interface on the service.

Return Values

S_OK

The service was successfully created or retrieved.

E_OUTOFMEMORY

There is not enough memory to create the service.

E_UNEXPECTED

An unknown error occurred.

E_NOINTERFACE

The service exists but the requested interface is not provided by the service.

SVC_E_UNKNOWNSERVICE

The service identified with *guidService* is not recognized.

Note to Callers

The caller is responsible for releasing the interface pointer when it is no longer needed.

Note to Implementers

Because there is only one method in this interface, E_NOTIMPL is not a valid return code.

ISimpleFrameSite Quick Info

The **ISimpleFrameSite** interface supports simple frame controls that act as simple containers for other nested controls. Some controls merely contain other controls. In such cases, the simple control container, called a simple frame, doesn't have to implement all container requirements. It can delegate most of the interface calls from its contained controls to the outer container that manages the simple frame. To support what are called simple frame controls, a container implements this interface as well as other site interfaces such as **IOleControlSite**.

An example of a simple frame control is a group box that only needs to capture a few keystrokes for its contained controls to implement the correct tab or arrow key behavior, but does not want to handle every other message. Through the two methods of this interface, the simple frame control passes messages to its control site both before and after its own processing. If that site is itself a simple frame, it can continue to pass messages up the chain.

When to Implement

Implement this interface on a control object to support nested controls without requiring the control to itself be a full container.

When to Use

Use this interface to pass messages to a container for processing.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

ISimpleFrameSite Methods	Description
PreMessageFilter	Sends the simple frame site a message that is received by a control's own window before the control itself does any processing.
PostMessageFilter	Sends the simple frame site a message that is received by a control's own window after the control does its own processing.

See Also

IOleControl, IOleControlSite

ISimpleFrameSite::PostMessageFilter Quick Info

Sends the simple frame site a message that is received by a control's own window after both **ISimpleFrameSite::PreMessageFilter** and the control have had a chance to process the message.

```
HRESULT PostMessageFilter(
  HWND hWnd ,          //Handle of window receiving message
  UINT msg ,           //Received message
  WPARAM wp ,          //WPARAM of message
  LPARAM lp ,          //LPARAM of message
  LRESULT* plResult ,  //Pointer to variable to receive result
  DWORD dwCookie       //Token returned by PreMessageFilter
);
```

Parameters

hWnd

[in] Handle of the control window receiving the message.

msg

[in] Message received by the simple frame site.

wp

[in] The **WPARAM** of the message.

lp

[in] The **LPARAM** of the message.

plResult

[out] Pointer to the variable that receives the result of the message processing.

dwCookie

[in] The **DWORD** value that was returned by **ISimpleFrameSite::PreMessageFilter** through its *pdwCookie* parameter.

Return Values

S_OK

The site processed the message.

S_FALSE

The site did not process the message.

E_NOTIMPL

The site does not filter any messages.

See Also

ISimpleFrameSite:PreMessageFilter

ISimpleFrameSite::PreMessageFilter Quick Info

Provides a site with the opportunity to process a message that is received by a control's own window before the control itself does any processing.

```
HRESULT PreMessageFilter(
  HWND hWnd ,          //Handle of window receiving message
  UINT msg ,           //Received message
  WPARAM wp ,          //WPARAM of message
  LPARAM lp ,          //LPARAM of message
  LRESULT* plResult ,  //Pointer to variable to receive result of message processing
  DWORD* pdwCookie     //Pointer to a variable used later
);
```

Parameters

hWnd

[in] Handle of the control window receiving the message.

msg

[in] Message received by the simple frame site.

wp

[in] The **WPARAM** of the message.

lp

[in] The **LPARAM** of the message.

plResult

[out] Pointer to the address of the result variable to receive the result of the message processing.

pdwCookie

[out] Pointer to the **DWORD** variable that will be passed to **PostMessageFilter** if it is called later. This parameter should only contain allocated data if this method returns S_OK so it will also receive a call to **PostMessageFilter** which can free the allocation. The caller is not in any way responsible for anything returned in this parameter.

Return Values

S_OK

The simple frame site will not use the message in this filter so more processing can take place.

S_FALSE

The site has processed the message and no further processing should occur.

E_NOTIMPL

The site does not do any message filtering, indicating that **PostMessageFilter** need not be called later.

E_POINTER

The addresses in *plResult*or *pdwCookie* are not valid.

Remarks

Successful return values indicate whether the site wishes to allow further processing. S_OK indicates further processing, whereas S_FALSE means do not process further. S_OK also indicates that the control must later call **ISimpleFrameSite::PostMessageFilter**.

See Also

ISimpleFrameSite::PostMessageFilter

ISpecifyPropertyPages Quick Info

The **ISpecifyPropertyPage** interface indicates that an object supports property pages. OLE property pages enable an object to display its properties in a tabbed dialog box known as a property sheet. An end user can then view and change the object's properties. An object can display its property pages independent of its client, or the client can manage the display of property pages from a number of contained objects in a single property sheet. Property pages also provide a means for notifying a client of changes in an object's properties.

A property page object manages a particular page within a property sheet. A property page implements at least **IPropertyPage** and can optionally implement **IPropertyPage2** if selection of a specific property is supported.

An object specifies its support for property pages by implementing **ISpecifyPropertyPages**. Through this interface the caller can obtain a list of CLSIDs identifying the specific property pages that the object supports. If the object specifies a property page CLSID, the object must be able to receive property changes from the property page.

When to Implement

Implement this interface on an object to indicate support for a property sheet and at least one property page.

When to Use

Use this interface to obtain a list of property page CLSIDs that this object supports. The CLSID list can be later passed to **OleCreatePropertyFrame** or **OleCreatePropertyFrameIndirect** to invoke a property sheet. If a caller wants to display a property sheet for multiple objects, it must first obtain the CLSID list for each object, then create a list containing only the intersection of the set of CLSID in each separate list. In other words, whoever invokes a property sheet for any number of objects must guarantee that each property page CLSID was specified by all objects for which the sheet is being displayed. This avoids the possibility that a property page is displayed for an object that doesn't understand that page; if this were allowed, it would result in problems when the page sent unknown and unexpected information to an object.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

ISpecifyPropertyPages Methods	Description
GetPages	Fills an array of CLSIDs for each property page that can be displayed in this object's property sheet.

See Also

IPerPropertyBrowsing, IPropertyPage, IPropertyPage2, IPropertyPageSite, OleCreatePropertyFrame, OleCreatePropertyFrameIndirect

ISpecifyPropertyPages::GetPages Quick Info

Fills a counted array of GUID values where each GUID specifies the CLSID of each property page that can be displayed in the property sheet for this object.

```
HRESULT GetPages(

  CAUUID *pPages      //Pointer to structure
);
```

Parameters

pPages

[out] Pointer to a caller-allocated **CAUUID** structure that must be initialized and filled before returning. The *pElems* field in the **CAUUID** structure is allocated by the callee with **CoTaskMemAlloc** and freed by the caller with **CoTaskMemFree**.

Return Values

This method supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The array was filled successfully.

E_POINTER

The address in *pPages* is not valid. For example, it may be NULL.

Remarks

The **CAUUID** structure is caller-allocated, but is not initialized by the caller. The **ISpecifyPropertyPages::GetPages** method fills the *cElements* field in the **CAUUID** structure**.** This method also allocates memory for the array pointed to by the *pElems* field in **CAUUID** using **CoTaskMemAlloc**. Then, it fills the newly allocated array. After this method returns successfully, the structure contains a counted array of UUIDs, each UUID specifying a property page CLSID.

Notes to Callers

The caller must release the memory pointed to by the *pElems* field of **CAUUID,** using **CoTaskMemFree** when it is no longer needed.

Notes to Implementers

E_NOTIMPL is not allowed as a return value since an object with no property pages should not expose the **ISpecifyPropertyPages** interface at all.

See Also

CAUUID, **CoTaskMemAlloc**, **CoTaskMemFree**, **OleCreatePropertyFrame**, **OleCreatePropertyFrameIndirect**

IStdMarshalInfo Quick Info

The **IStdMarshalInfo** interface returns the CLSID identifying the handler to be used in the destination process during standard marshaling.

An object that uses OLE's default implementation of **IMarshal** does not provide its own proxy but, by implementing **IStdMarshalInfo**, can nevertheless specify a handler to be loaded in the client process. Such a handler would typically handle certain requests in-process and use OLE's default marshaling to delegate others back to the original object.

To create an instance of an object in some client process, COM must first determine whether the object uses default marshaling or its own implementation. If the object uses default marshaling, COM then queries the object to determine whether it uses a special handler or, simply, OLE's default proxy. To get the CLSID of the handler to be loaded, COM queries the object for the **IStdMarshalInfo** interface and then the **IPersist** interface. If neither interface is supported, a standard handler is used.

When to Implement

If you are writing a server application that supports class emulation (that is, if your server can manipulate objects of another type in response to the Activate As option in the Convert dialog box), you must implement the **IStdMarshalInfo** interface in order to return the CLSID of the handler to be used for the object.

Note that your handler must aggregate the default handler.

When to Use

You typically don't call this interface yourself. COM queries for this interface when performing standard marshaling.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IStdMarshalInfo Method	Description
GetClassForHandler	Obtains the class identifier of the object handler in the destination process.

See Also

IMarshal

IStdMarshalInfo::GetClassForHandler Quick Info

Retrieves the CLSID of the object handler to be used in the destination process during standard marshaling.

```
HRESULT GetClassForHandler(
    DWORD dwDestContext,    //Destination process
    void * pvDestContext,   //Reserved
    CLSID * pClsid          //Pointer to the CLSID
);
```

Parameters

dwDestContext

[in] Destination context, that is, the process in which the unmarshaling will be done. The legal values for *dwDestContext* are taken from the enumeration **MSHCTX**. For information on the **MSHCTX** enumeration, see the "Data Structures" section.

pvDestContext

[in] Reserved for future use; must be NULL.

pClsid

[out] Pointer to the handler's CLSID.

Return Values

This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The CLSID was retrieved successfully.

Remarks

Notes to Implementers

Your implementation of **IStdMarshalInfo::GetClassForHandler** must return your own CLSID. This allows an object created by a different server to behave as one your server created.

IStorage

Quick Info

The **IStorage** interface supports the creation and management of structured storage objects. Structured storage allows hierarchical storage of information within a single file, and is often referred to as "a file system within a file". Elements of a structured storage object are storages and streams. Storages are analogous to directories, and streams are analogous to files. Within a structured storage there will be a primary storage object that may contain substorages, possibly nested, and streams. Storages provide the structure of the object, and streams contain the data, which is manipulated through the **IStream** interface.

The **IStorage** interface provides methods for creating and managing the root storage object, child storage objects, and stream objects. These methods can create, open, enumerate, move, copy, rename, or delete the elements in the storage object.

An application must release its **IStorage** pointers when it is done with the storage object to deallocate memory used. There are also methods for changing the date and time of an element.

There are a number of different modes in which a storage object and its elements can be opened, determined by setting values from the **STGM** enumeration. One aspect of this is how changes are committed. You can set direct mode, in which in which changes to an object are immediately written to it, or transacted mode, in which changes are written to a buffer until explicitly committed. The **IStorage** interface provides methods for committing changes and reverting to the last-committed version. Other storage modes set, for example, a stream can be opened in read only mode or read/write mode. For more information, refer to the **STGM** enumeration.

Other methods provide a means to gain access to information about a storage object and its elements through the **STATSTG** structure.

When to Implement

Generally, you would not implement this interface unless you were defining a new storage scheme for your system. OLE provides a compound file implementation of the **IStorage** interface that supports transacted access. OLE provides a set of helper APIs to facilitate using the compound file implementation of storage objects. Refer to **IStorage - Compound File Implementation**.

When to Use

Call the methods of **IStorage** to manage substorages or streams within the current storage. This management includes creating, opening, or destroying sub-storages or streams, as well as managing aspects such as time stamps, names, etc. You can also commit changes or revert to previous version for storages opened in transacted mode. The methods of **IStorage** do not include means to read and write data—this is reserved for **IStream**, which manages the actual data. While the **IStorage** and **IStream** interfaces are used to manipulate the storage object and its elements, the **IPersistStorage** interface contains methods that are called to serialize the storage object and its elements to a disk file.

Methods VTable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IStorage Methods	**Description**
CreateStream	Creates and opens a stream object with the specified name contained

	in this storage object.
OpenStream	Opens an existing stream object within this storage object using the specified access permissions in *grfMode*.
CreateStorage	Creates and opens a new storage object within this storage object.
OpenStorage	Opens an existing storage object with the specified name according to the specified access mode.
CopyTo	Copies the entire contents of this open storage object into another storage object. The layout of the destination storage object may differ.
MoveElementTo	Copies or moves a substorage or stream from this storage object to another storage object.
Commit	Reflects changes for a transacted storage object to the parent level.
Revert	Discards all changes that have been made to to the storage object since the last commit operation.
EnumElements	Returns an enumerator object that can be used to enumerate the storage and stream objects contained within this storage object.
DestroyElement	Removes the specified storage or stream from this storage object.
RenameElement	Renames the specified storage or stream in this storage object.
SetElementTimes	Sets the modification, access, and creation times of the indicated storage element, if supported by the underlying file system.
SetClass	Assigns the specified CLSID to this storage object.
SetStateBits	Stores up to 32 bits of state information in this storage object.
Stat	Returns the **STATSTG** structure for this open storage object.

IStorage::Commit Quick Info

Ensures that any changes made to a storage object open in transacted mode are reflected in the parent storage; for a root storage, reflects the changes in the actual device, for example, a file on disk. For a root storage object opened in direct mode, this method has no effect except to flush all memory buffers to the disk. For non-root storage objects in direct mode, this method has no effect.

```
HRESULT Commit(
    DWORD grfCommitFlags        //Specifies how changes are to be committed
);
```

Parameter

grfCommitFlags

[in] Controls how the changes are committed to the storage object. See the **STGC** enumeration for a definition of these values.

Return Values

S_OK

Changes to the storage object were successfully committed to the parent level.

E_PENDING

Asynchronous Storage only: Part or all of the data to be committed is currently unavailable. For more information, see **IFillLockBytes** and Asynchronous Storage.

STG_E_INVALIDFLAG

The value for the *grfCommitFlags* parameter is not valid.

STG_E_INVALIDPARAMETER

One of the parameters was not valid.

STG_E_NOTCURRENT

Another open instance of the storage object has committed changes. Thus, the current commit operation may overwrite previous changes.

STG_E_MEDIUMFULL

No space left on device to commit.

STG_E_TOOMANYOPENFILES

The commit operation could not be completed because there are too many open files.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

Remarks

IStorage::Commit makes permanent changes to a storage object that is in transacted mode, in which changes are accumulated in a buffer, and not reflected in the storage object until there is a call to this method. The alternative is to open an object in direct mode, in which changes are immediately reflected in the storage object and so does not require a commit operation. Calling this method on a storage opened in direct mode has no effect, unless it is a root storage, in which case it ensures that changes in memory buffers are written to the underlying storage device.

The commit operation publishes the current changes in this storage object and its children to the next level up in the storage hierarchy. To undo current changes before committing them, call **IStorage::Revert** to roll back to the last-committed version.

Calling **IStorage::Commit** has no effect on currently-opened nested elements of this storage object. They are still valid and can be used. However, the **IStorage::Commit** method does not automatically commit changes to these nested elements. The commit operation publishes only known changes to the next higher level of the storage hierarchy. Thus, transactions to nested levels must be committed to this storage object before they can be committed to higher levels.

In commit operations, you need to take steps to ensure that data is protected during the commit process:

- When committing changes to root storage objects, the caller must check the return value to determine whether the operation has been completed successfully, and if not, that the old committed contents of the **IStorage** are still intact and can be restored.
- If this storage object was opened with some of its items excluded, then the caller is responsible for rewriting them before calling commit. Write mode is required on the storage opening for the commit to succeed.
- Unless prohibiting multiple simultaneous writers on the same storage object, an application calling this method should specify at least STGC_ONLYIFCURRENT in the *grfCommitFlags* parameter to prevent the changes made by one writer from inadvertently overwriting the changes made by another.

See Also

IStorage - Compound File Implementation, **STGC**, **IStorage::Revert**

IStorage::CopyTo Quick Info

Copies the entire contents of an open storage object to another storage object.

```
HRESULT CopyTo(
    DWORD ciidExclude,              //Number of elements in rgiidExclude
    IID const * rgiidExclude,       //Array of interface identifiers (IIDs)
    SNB snbExclude,                 //Points to a block of stream names in the storage object
    IStorage * pstgDest             //Points to destination storage object
);
```

Parameters

ciidExclude

[in] The number of elements in the array pointed to by *rgiidExclude*. If *rgiidExclude* is NULL, then *ciidExclude* is ignored.

rgiidExclude

[in] An array of interface identifiers that either the caller knows about and does not want to be copied or that the storage object does not support but whose state the caller will later explicitly copy. The array can include **IStorage**, indicating that only stream objects are to be copied, and **IStream**, indicating that only storage objects are to be copied. An array length of zero indicates that only the state exposed by the **IStorage** object is to be copied; all other interfaces on the object are to be ignored. Passing NULL indicates that all interfaces on the object are to be copied.

snbExclude

[in] A string name block (refer to SNB) that specifies a block of storage or stream objects that are not to be copied to the destination. These elements are not created at the destination. If IID_IStorage is in the *rgiidExclude* array, this parameter is ignored. This parameter may be NULL.

pstgDest

[in] Points to the open storage object into which this storage object is to be copied. The destination storage object can be a different implementation of the **IStorage** interface from the source storage object. Thus, **IStorage::CopyTo** can only use publicly available methods of the destination storage object. If *pstgDest* is open in transacted mode, it can be reverted by calling its **IStorage::Revert** method.

Return Values

S_OK

The storage object was successfully copied.

E_PENDING

Asynchronous Storage only: Part or all of the data to be copied is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The destination storage object is a child of the source storage object.

STG_E_INSUFFICIENTMEMORY

The copy was not completed due to a lack of memory.

STG_E_INVALIDPOINTER

The pointer specified for the storage object was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_TOOMANYOPENFILES

The copy was not completed because there are too many open files.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_MEDIUMFULL

The copy was not completed because the storage medium is full.

Remarks

This method merges elements contained in the source storage object with those already present in the destination. The layout of the destination storage object may differ from the source storage object.

The copy process is recursive, invoking **IStorage::CopyTo** and **IStream::CopyTo** on the elements nested inside the source.

When copying a stream on top of an existing stream with the same name, the existing stream is first removed and then replaced with the source stream. When copying a storage on top of an existing storage with the same name, the existing storage is not removed. As a result,, after the copy operation, the destination **IStorage** contains older elements, unless they were replaced by newer ones with the same names.

A storage object may expose interfaces other than **IStorage**, including **IRootStorage**, **IPropertyStorage**, or **IPropertySetStorage**. The *rgiidExclude* parameter provides a way to exclude any or all of these additional interfaces from the copy operation.

A caller with a newer or more efficient copy of an existing substorage or stream object may want to exclude the current versions of these objects from the copy operation. The *snbExclude* and *rgiidExclude* parameters provide two different ways of excluding a storage objects existing storages or streams.

Note to Callers

The most common way to use this method is to copy everything possible from the source to the destination, as in most Full Save and SaveAs operations. The following example illustrates this call:

```
pstg->CopyTo(0, Null, Null, pstgDest)
```

See Also

IStorage - Compound File Implementation, **IStorage::MoveElementTo**, **IStorage::Revert**

IStorage::CreateStorage Quick Info

Creates and opens a new storage object nested within this storage object.

```
HRESULT CreateStorage(
  const WCHAR * pwcsName,        //Points to the name of the new storage object
  DWORD grfMode,                 //Access mode for the new storage object
  DWORD reserved1,               //Reserved; must be zero
  DWORD reserved2,               //Reserved; must be zero
  IStorage ** ppstg              //Points to new storage object
);
```

Parameters

pwcsName

[in] Points to a wide character string that contains the name of the newly created storage object. This name can be used later to reopen the storage object.

grfMode

[in] Specifies the access mode to use when opening the newly created storage object. See the **STGM** enumeration values for descriptions of the possible values.

reserved1

[in] Reserved for future use; must be zero.

reserved2

[in] Reserved for future use; must be zero.

ppstg

[out] When successful, points to the location of the **IStorage** pointer to the newly-created storage object. This parameter is set to NULL if an error occurs.

Return Values

S_OK

The storage object was created successfully.

E_PENDING

Asynchronous Storage only: Part or all of the necessary data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

Insufficient permissions to create storage object.

STG_E_FILEALREADYEXISTS

The name specified for the storage object already exists in the storage object and the *grfmode* flag includes the flag STGM_FAILIFTHERE.

STG_E_INSUFFICIENTMEMORY

The storage object was not created due to a lack of memory.

STG_E_INVALIDFLAG

The value specified for the *grfMode* flag is not a valid **STGM** enumeration value.

STG_E_INVALIDFUNCTION

The specified combination of *grfMode* flags is not supported.

STG_E_INVALIDNAME

Invalid value for *pwcsName*.

STG_E_INVALIDPOINTER

The pointer specified for the storage object was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_TOOMANYOPENFILES

The storage object was not created because there are too many open files.

STG_S_CONVERTED

The existing stream with the specified name was replaced with a new storage object containing a single stream called CONTENTS. The new storage object will be added.

Remarks

If a storage with the name specified in the *pwcsName* parameter already exists within the parent storage object, and the *grfMode* parameter includes the STGM_CREATE flag, the existing storage is replaced by the new one. If the *grfMode* parameter includes the STGM_CONVERT flag, the existing element is converted to a stream object named CONTENTS and the new storage object is created containing the CONTENTS stream object. The destruction of the old element and the creation of the new storage object are both subject to the transaction mode on the parent storage object.

If a storage object with the same name already exists and *grfMode* is set to STGM_FAILIFTHERE, this method fails with the return value STG_E_FILEALREADYEXISTS.

See Also

IStorage - Compound File Implementation, **IStorage::OpenStorage**

IStorage::CreateStream Quick Info

Creates and opens a stream object with the specified name contained in this storage object. All elements within a storage object – both streams and other storage objects – are kept in the same name space.

```
HRESULT CreateStream(
  const WCHAR * pwcsName,        //Points to the name of the new stream
  DWORD grfMode,                 //Access mode for the new stream
  DWORD reserved1,               //Reserved; must be zero
  DWORD reserved2,               //Reserved; must be zero
  IStream ** ppstm               //Points to new stream object
);
```

Parameters

pwcsName

[in] Points to a wide character string that contains the name of the newly created stream. This name can be used later to open or reopen the stream.

grfMode

[in] Specifies the access mode to use when opening the newly created stream. See the **STGM** enumeration values for descriptions of the possible values.

reserved1

[in] Reserved for future use; must be zero.

reserved2

[in] Reserved for future use; must be zero.

ppstm

[out] On return, points to the location of the new **IStream** interface pointer. This is only valid if the operation is successful. When an error occurs, this parameter is set to NULL.

Return Values

S_OK

The new stream was successfully created

E_PENDING

Asynchronous Storage only: Part or all of the necessary data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

Insufficient permissions to create stream.

STG_E_FILEALREADYEXISTS

The name specified for the stream already exists in the storage object and the *grfmode* flag includes the flag STGM_FAILIFTHERE.

STG_E_INSUFFICIENTMEMORY

The stream was not created due to a lack of memory.

STG_E_INVALIDFLAG

The value specified for the *grfMode* flag is not a valid **STGM** enumeration value.

STG_E_INVALIDFUNCTION

The specified combination of *grfMode* flags is not supported. For example, if this method is called without the STGM_SHARE_EXCLUSIVE flag.

STG_E_INVALIDNAME

Invalid value for *pwcsName*.

STG_E_INVALIDPOINTER

The pointer specified for the stream object was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_TOOMANYOPENFILES

The stream was not created because there are too many open files.

Remarks

If a stream with the name specified in the *pwcsName* parameter already exists and the *grfMode* parameter includes the STGM_CREATE flag, the existing stream is replaced by a newly created one. Both the destruction of the old stream and the creation of the new stream object are subject to the transaction mode on the parent storage object.

If the stream already exists and *grfMode* is set to STGM_FAILIFTHERE, this method fails with the return value STG_E_FILEALREADYEXISTS.

See Also

IStorage - Compound File Implementation, **IStorage::OpenStream**, **IStream**

IStorage::DestroyElement Quick Info

Removes the specified storage or stream from this storage object.

```
HRESULT DestroyElement(
  wchar * pwcsName                    //Points to the name of the element to be removed
);
```

Parameter

pwcsName

[in] Points to a wide character string that contains the name of the storage or stream to be removed.

Return Values

S_OK

The element was successfully removed.

E_PENDING

Asynchronous Storage only: Part or all of the element's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for removing the element.

STG_E_FILENOTFOUND

The element with the specified name does not exist.

STG_E_INSUFFICIENTMEMORY

The element was not removed due to a lack of memory.

STG_E_INVALIDNAME

Invalid value for *pwcsName*.

STG_E_INVALIDPOINTER

The pointer specified for the element was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_TOOMANYOPENFILES

The element was not removed because there are too many open files.

Remarks

The **DestroyElement** method deletes a substorage or stream from the current storage object. After a successful call to **DestroyElement**, any open instance of the destroyed element from the parent storage becomes invalid.

If a storage object is opened in transacted mode, destruction of an element requires that the call to

DestroyElement be followed by a call to **IStorage::Commit**.

See Also

IStorage - Compound File Implementation

IStorage::EnumElements Quick Info

Retrieves a pointer to an enumerator object that can be used to enumerate the storage and stream objects contained within this storage object.

```
HRESULT EnumElements(
  DWORD reserved1,                //Reserved; must be zero
  void * reserved2,               //Reserved; must be NULL
  DWORD reserved3,                //Reserved; must be zero
  IEnumSTATSTG ** ppenum          //Indirect pointer to IEnumSTATSTG
);
```

Parameters

reserved1

[in] Reserved for future use; must be zero.

reserved2

[in] Reserved for future use; must be NULL.

reserved3

[in] Reserved for future use; must be zero.

ppenum

[out] When successful, points to the location of an **IEnumSTATSTG** pointer to new enumerator object.

Return Values

S_OK

The enumerator object was successfully returned.

E_PENDING

Asynchronous Storage only: Part or all of the element's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_INSUFFICIENTMEMORY

The enumerator object could not be created due to lack of memory.

STG_E_INVALIDPARAMETER

One of the parameters was not valid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

Remarks

The enumerator object returned by this method implements the **IEnumSTATSTG** interface, one of the standard enumerator interfaces that contain the **Next**, **Reset**, **Clone**, and **Skip** methods. **IEnumSTATSTG** enumerates the data stored in an array of **STATSTG** structures.

The storage object must be open in read mode to allow the enumeration of its elements.

The order in which the elements are enumerated and whether the enumerator is a snapshot or always reflects the current state of the storage object, and depends on the **IStorage** implementation.

See Also

IStorage - Compound File Implementation, **IEnumXXXX**, **IEnumSTATSTG**, **STATSTG**

IStorage::MoveElementTo Quick Info

Copies or moves a substorage or stream from this storage object to another storage object.

```
HRESULT MoveElementTo(
  const WCHAR * pwcsName,      //Name of the element to be moved
  IStorage * pstgDest,         //Points to destination storage object
  LPWSTR pwcsNewName,          //Points to new name of element in destination
  DWORD grfFlags               //Specifies a copy or a move
);
```

Parameters

pwcsName

[in] Points to a wide character string that contains the name of the element in this storage object to be moved or copied.

pstgDest

[in] **IStorage** pointer to the destination storage object.

pwcsNewName

[in] Points to a wide character string that contains the new name for the element in its new storage object.

grfFlags

[in] Specifies whether the operation should be a move (STGMOVE_MOVE) or a copy (STGMOVE_COPY). See the **STGMOVE** enumeration.

Return Values

S_OK

The storage object was successfully copied or moved.

E_PENDING

Asynchronous Storage only: Part or all of the element's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The destination storage object is a child of the source storage object.

STG_E_FILENOTFOUND

The element with the specified name does not exist.

STG_E_FILEALREADYEXISTS

The specified file already exists.

STG_E_INSUFFICIENTMEMORY

The copy or move was not completed due to a lack of memory.

STG_E_INVALIDFLAG

The value for the *grfFlags* parameter is not valid.

STG_E_INVALIDNAME

Invalid value for *pwcsName*.

STG_E_INVALIDPOINTER

The pointer specified for the storage object was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_TOOMANYOPENFILES

The copy or move was not completed because there are too many open files.

Remarks

The **IStorage::MoveElementTo** method is typically the same as invoking the **IStorage::CopyTo** method on the indicated element and then removing the source element. In this case, the **MoveElementTo** method uses only the publicly available functions of the destination storage object to carry out the move.

If the source and destination storage objects have special knowledge about each other's implementation (they could, for example, be different instances of the same implementation), this method can be implemented more efficiently.

Before calling this method, the element to be moved must be closed, and the destination storage must be open.

See Also

IStorage - Compound File Implementation, **STGMOVE**, **IStorage::CopyTo**

IStorage::OpenStorage Quick Info

Opens an existing storage object with the specified name in the specified access mode.

```
HRESULT OpenStorage(
  const WCHAR * pwcsName,          //Points to the name of the storage object to open
  IStorage * pstgPriority,         //Points to previous opening of the storage object
  DWORD grfMode,                   //Access mode for the new storage object
  SNB snbExclude,                  //Points to a block of stream names in the storage object
  DWORD reserved,                  //Reserved; must be zero
  IStorage ** ppstg                //Points to opened storage object
);
```

Parameters

pwcsName

[in] Points to a wide character string that contains the name of the storage object to open. It is ignored if *pstgPriority* is non-NULL.

pstgPriority

[in] If the *pstgPriority* parameter is not NULL, it is an **IStorage** pointer to a previous opening of an element of the storage object, usually one that was opened in priority mode. The storage object should be closed and re-opened according to *grfMode*. When the **IStorage::OpenStorage** method returns, *pstgPriority* is no longer valid. Use the value supplied in the *ppstg* parameter. If the *pstgPriority* parameter is NULL, it is ignored.

grfMode

[in] Specifies the access mode to use when opening the storage object. See the **STGM** enumeration values for descriptions of the possible values. Whatever other modes you may choose, you must at least specify STGM_SHARE_EXCLUSIVE when calling this method.

snbExclude

[in] Must be NULL. A non-NULL value will return STG_E_INVALIDPARAMETER.

reserved

[in] Reserved for future use; must be zero.

ppstg

[out] When the operation is successful, points to the location of an **IStorage** pointer to the opened storage object. This parameter is set to NULL if an error occurs.

Return Values

S_OK

The storage object was opened successfully.

E_PENDING

Asynchronous Storage only: Part or all of the storage's data is currently unavailable. For more

information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

Insufficient permissions to open storage object.

STG_E_FILENOTFOUND

The storage object with the specified name does not exist.

STG_E_INSUFFICIENTMEMORY

The storage object was not opened due to a lack of memory.

STG_E_INVALIDFLAG

The value specified for the *grfMode* flag is not a valid **STGM** enumeration value.

STG_E_INVALIDFUNCTION

The specified combination of *grfMode* flags is not supported.

STG_E_INVALIDNAME

Invalid value for *pwcsName*.

STG_E_INVALIDPOINTER

The pointer specified for the storage object was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_TOOMANYOPENFILES

The storage object was not created because there are too many open files.

STG_S_CONVERTED

The existing stream with the specified name was replaced with a new storage object containing a single stream called CONTENTS. In direct mode, the new storage is immediately written to disk. In transacted mode, the new storage is written to a temporary storage in memory and later written to disk when it is committed.

Remarks

Storage objects can be opened with STGM_DELETEONRELEASE, in which case the object is destroyed when it receives its final release. This is useful for creating temporary storage objects.

See Also

IStorage - Compound File Implementation, **IStorage::CreateStorage**

IStorage::OpenStream Quick Info

Opens an existing stream object within this storage object in the specified access mode.

```
HRESULT OpenStream(
  const WCHAR * pwcsName,      //Points to name of stream to open
  void * reserved1,            //Reserved; must be NULL
  DWORD grfMode,               //Access mode for the new stream
  DWORD reserved2,             //Reserved; must be zero
  IStream ** ppstm             //Indirect pointer to opened stream object
);
```

Parameters

pwcsName

[in] Points to a wide character string that contains the name of the stream to open.

reserved1

[in] Reserved for future use; must be NULL.

grfMode

[in] Specifies the access mode to be assigned to the open stream. See the **STGM** enumeration values for descriptions of the possible values. . Whatever other modes you may choose, you must at least specify STGM_SHARE_EXCLUSIVE when calling this method.

reserved2

[in] Reserved for future use; must be zero.

ppstm

[out] On successful return, points to the location of an **IStream** pointer to the newly-opened stream object. This parameter is set to NULL if an error occurs.

Return Values

S_OK

The stream was successfully opened.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

Insufficient permissions to open stream.

STG_E_FILENOTFOUND

The stream with specified name does not exist.

STG_E_INSUFFICIENTMEMORY

The stream was not opened due to a lack of memory.

STG_E_INVALIDFLAG

The value specified for the *grfMode* flag is not a valid **STGM** enumeration value.

STG_E_INVALIDFUNCTION

The specified combination of *grfMode* flags is not supported. For example, if this method is called without the STGM_SHARE_EXCLUSIVE flag.

STG_E_INVALIDNAME

Invalid value for *pwcsName*.

STG_E_INVALIDPOINTER

The pointer specified for the stream object was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_TOOMANYOPENFILES

The stream was not opened because there are too many open files.

Remarks

IStorage::OpenStream opens an existing stream object within this storage object in the access mode specified in *grfMode*. There are restrictions on the permissions that can be given in *grfMode*. For example, the permissions on this storage object restrict the permissions on its streams. In general, access restrictions on streams should be stricter than those on their parent storages. Compound-file streams must be opened with STGM_SHARE_EXCLUSIVE.

See Also

IStorage - Compound File Implementation, **IStorage::CreateStream**, **IStream**

IStorage::RenameElement Quick Info

Renames the specified substorage or stream in this storage object.

```
HRESULT RenameElement(
  const WCHAR * pwcsOldName,      //Points to the name of the element to be changed
  const WCHAR * pwcsNewName       //Points to the new name for the specified element
);
```

Parameters

pwcsOldName

[in] Points to a wide character string that contains the name of the substorage or stream to be changed.

pwcsNewName

[in] Points to a wide character string that contains the new name for the specified sustorage or stream.

Return Values

S_OK

The element was successfully renamed.

E_PENDING

Asynchronous Storage only: Part or all of the element's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for renaming the element.

STG_E_FILENOTFOUND

The element with the specified old name does not exist.

STG_E_FILEALREADYEXISTS

The element specified by the new name already exists.

STG_E_INSUFFICIENTMEMORY

The element was not renamed due to a lack of memory.

STG_E_INVALIDNAME

Invalid value for one of the names.

STG_E_INVALIDPOINTER

The pointer specified for the element was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

STG_E_TOOMANYOPENFILES

The element was not renamed because there are too many open files.

Remarks

IStorage::RenameElement renames the specified substorage or stream in this storage object. An element in a storage object cannot be renamed while it is open. The rename operation is subject to committing the changes if the storage is open in transacted mode.

The **IStorage::RenameElement** method is not guaranteed to work in low memory with storage objects open in transacted mode. It may work in direct mode.

See Also

IStorage - Compound File Implementation

IStorage::Revert Quick Info

Discards all changes that have been made to the storage object since the last commit.

HRESULT Revert(*void***);**

Return Values

S_OK

The revert operation was successful.

E_PENDING

Asynchronous Storage only: Part or all of the storage's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_INSUFFICIENTMEMORY

The revert operation could not be completed due to a lack of memory.

STG_E_TOOMANYOPENFILES

The revert operation could not be completed because there are too many open files.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

Remarks

For storage objects opened in transacted mode, the **IStorage::Revert** method discards any uncommitted changes to this storage object or changes that have been committed to this storage object from nested elements.

After this method returns, any existing elements (substorages or streams) that were opened from the reverted storage object are invalid and can no longer be used. Specifying these reverted elements in any call except **IStorage::Release** returns the error STG_E_REVERTED

This method has no effect on storage objects opened in direct mode.

See Also

IStorage - Compound File Implementation, **IStorage::Commit**

IStorage::SetClass Quick Info

Assigns the specified CLSID to this storage object.

```
HRESULT SetClass(
  REFCLSID clsid          //Class identifier to be assigned to the storage object
);
```

Parameter

clsid

[in] The class identifier (CLSID) that is to be associated with the storage object.

Return Values

S_OK

The CLSID was successfully assigned.

E_PENDING

Asynchronous Storage only: Part or all of the storage's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for assigning a class identifier to the storage object.

STG_E_MEDIUMFULL

Not enough space was left on device to complete the operation.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

Remarks

When first created, a storage object has an associated CLSID of CLSID_NULL. Call this method to assign a CLSID to the storage object.

Call the **IStorage::Stat** method to retrieve the current CLSID of a storage object.

See Also

IStorage - Compound File Implementation, **IStorage::Stat**

IStorage::SetElementTimes Quick Info

Sets the modification, access, and creation times of the specified storage element, if supported by the underlying file system.

```
HRESULT SetElementTimes(
  const WCHAR * pwcsName,        //Points to name of element to be changed
  FILETIME const * pctime,       //New creation time for element, or NULL
  FILETIME const * patime,       //New access time for element, or NULL
  FILETIME const * pmtime        //New modification time for element, or NULL
);
```

Parameters

pwcsName

[in] The name of the storage object element whose times are to be modified. If NULL, the time is set on the root storage rather than one of its elements.

pctime

[in] Either the new creation time for the element or NULL if the creation time is not to be modified.

patime

[in] Either the new access time for the element or NULL if the access time is not to be modified.

pmtime

[in] Either the new modification time for the element or NULL if the modification time is not to be modified.

Return Values

S_OK

The time values were successfully set.

E_PENDING

Asynchronous Storage only: Part or all of the element's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for changing the element.

STG_E_FILENOTFOUND

The element with the specified name does not exist.

STG_E_INSUFFICIENTMEMORY

The element was not changed due to a lack of memory.

STG_E_INVALIDNAME

Invalid value for the element name.

STG_E_INVALIDPOINTER

The pointer specified for the element was invalid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

STG_E_TOOMANYOPENFILES

The element was not changed because there are too many open files.

STG_E_REVERTED

The storage object has been invalidated by a revert operation above it in the transaction tree.

Remarks

This method sets time statistics for the specified storage element within this storage object.

Not all file systems support all of the time values. This method sets those times that are supported and ignores the rest. Each of the time value parameters can be NULL; indicating that no modification should occur.

Call the **IStorage::Stat** method to retrieve these time values.

See Also

IStorage - Compound File Implementation, **IStorage::Stat**

IStorage::SetStateBits Quick Info

Stores up to 32 bits of state information in this storage object.

```
HRESULT SetStateBits(
  DWORD grfStateBits,      //Specifies new values of bits
  DWORD grfMask            //Specifies mask that indicates which bits are significant
);
```

Parameters

grfStateBits

[in] Specifies the new values of the bits to set. No legal values are defined for these bits; they are all reserved for future use and must not be used by applications.

grfMask

[in] A binary mask indicating which bits in *grfStateBits* are significant in this call.

Return Values

S_OK

The state information was successfully set.

E_PENDING

Asynchronous Storage only: Part or all of the storage's data is currently unavailable. For more information see **ILockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for changing this storage object.

STG_E_INVALIDFLAG

The value for the *grfStateBits* or *grfMask* parameters are not valid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

Remarks

This interface is reserved for future use. The values for the state bits are not currently defined.

See Also

IStorage - Compound File Implementation, **IStorage::Stat**

IStorage::Stat Quick Info

Retrieves the **STATSTG** structure for this open storage object.

```
HRESULT Stat(
  STATSTG * pstatstg,          //Location for STATSTG structure
  DWORD grfStatFlag            //Values taken from the STATFLAG enumeration
);
```

Parameters

pstatstg

[out] On return, points to a **STATSTG** structure where this method places information about the open storage object. This parameter is NULL if an error occurs.

grfStatFlag

[in] Specifies that some of the fields in the **STATSTG** structure are not returned, thus saving a memory allocation operation. Values are taken from the **STATFLAG** enumeration.

Return Values

S_OK

The **STATSTG** structure was successfully returned at the specified location.

E_PENDING

Asynchronous Storage only: Part or all of the storage's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for accessing statistics for this storage object.

STG_E_INSUFFICIENTMEMORY

The **STATSTG** structure was not returned due to a lack of memory.

STG_E_INVALIDFLAG

The value for the *grfStateFlag* parameter is not valid.

STG_E_INVALIDPARAMETER

One of the parameters was invalid.

Remarks

IStorage::Stat retrieves the STATSTG structure for the current storage. This structure contains statistical information about the storage. **IStorage::EnumElements** creates an enumerator object with the **IEnumSTATSTG** interface, though which you can enumerate the substorages and streams of a storage through the STATSTG structure of each.

See Also

IStorage - Compound File Implementation, **STATFLAG**, **STATSTG**, **IEnumSTATSTG**, **IStorage::SetClass**, **IStorage::SetElementTimes**, **IStorage::SetStateBits**

IStorage - Compound File Implementation

The compound file implementation of **IStorage** allows you to create and manage substorages and streams within a storage object residing in a compound file object. To create a compound file object and get an **IStorage** pointer, call the API function **StgCreateDocfile**. To open an existing compound file object and get its root **IStorage** pointer, call **StgOpenStorage**.

When to Use

Most applications use this implementation to to create and manage storages and streams.

Remarks

IStorage::CreateStream

Creates and opens a stream object with the specified name contained in this storage object. The OLE-provided compound file implementation of the **IStorage::CreateStream** method does not support the following behaviors:

- The STGM_DELETEONRELEASE flag is not supported.
- Transacted mode is not supported for stream objects.
- Opening the same stream more than once from the same storage is not supported. The STGM_SHARE_EXCLUSIVE flag must be specified.

IStorage::OpenStream

Opens an existing stream object within in this storage object using the specified access modes specified in the grfMode parameter. The OLE-provided compound file implementation of the **IStorage::OpenStream** method does not support the following behavior:

- The STGM_DELETEONRELEASE flag is not supported.
- Transacted mode is not supported for stream objects.
- Opening the same stream more than once from the same storage is not supported. The STGM_SHARE_EXCLUSIVE flag must be specified.

IStorage::CreateStorage

The OLE-provided compound file implementation of the **IStorage::CreateStorage** method does not support the STGM_DELETEONRELEASE flag. Specifying this flag causes the method to return STG_E_INVALIDFLAG.

IStorage::OpenStorage

Opens an existing storage object with the specified name in the specified access mode. The OLE-provided compound file implementation of the **IStorage::OpenStorage** method does not support the following behavior:

- The STGM_PRIORITY flag is not supported for non-root storages.
- Opening the same storage object more than once from the same parent storage is not supported. The STGM_SHARE_EXCLUSIVE flag must be specified.
- The STGM_DELETEONRELEASE flag is not supported. If this flag is specified, the function returns STG_E_INVALIDFUNCTION.

IStorage::CopyTo

Copies only the substorages and streams of this open storage object into another storage object. The *rgiidExclude* parameter can be set to IID_IStream to copy only substorages, or to IID_IStorage to copy only streams.

IStorage::MoveElementTo

Copies or moves a substorage or stream from this storage object to another storage object.

IStorage::Commit

Ensures that any changes made to a storage object open in transacted mode are reflected in the parent storage; for a root storage, reflects the changes in the actual device, for example, a file on disk. For a root storage object opened in direct mode, this method has no effect except to flush all memory buffers to the disk. For non-root storage objects in direct mode, this method has no effect.

The OLE-provided compound files implementation uses a two phase commit process unless STGC_OVERWRITE is specified in the *grfCommitFlags* parameter. This two-phase process ensures the robustness of data in case the commit operation fails. First, all new data is written to unused space in the underlying file. If necessary, new space is allocated to the file. Once this step has been successfully completed, a table in the file is updated using a single sector write to indicate that the new data is to be used in place of the old. The old data becomes free space to be used at the next commit. Thus, the old data is available and can be restored in case an error occurs when committing changes. If STGC_OVERWRITE is specified, a single phase commit operation is used.

IStorage::Revert

Discards all changes that have been made to the storage object since the last commit.

IStorage::EnumElements

Creates and retrieves a pointer to an enumerator object that can be used to enumerate the storage and stream objects contained within this storage object.The OLE-provided compound file implementation takes a snapshot.

IStorage::DestroyElement

Removes the specified element (substorage or stream) from this storage object.

IStorage::RenameElement

Renames the specified substorage or stream in this storage object.

IStorage::SetElementTimes

Sets the modification, access, and creation times of the specified storage element. The OLE-provided compound file implementation maintains modification and change times for internal storage objects. For root storage objects, whatever is supported by the underlying file system (or **ILockBytes**) is supported. The compound file implementation does not maintain any time stamps for internal streams. Unsupported time stamps are reported as zero, enabling the caller to test for support.

IStorage::SetClass

Assigns the specified CLSID to this storage object.

IStorage::SetStateBits

Stores up to 32 bits of state information in this storage object. The state set by this method is for external use only. The OLE-provided compound file implementation does not perform any action based on the state.

IStorage::Stat

Retrieves the **STATSTG** structure for this open storage object.

See Also

IStorage, **IStream**, **StgCreateDocfile**, **StgOpenStorage**, **IFillLockBytes**, **ILockBytes**, **IRootStorage**

IStream Quick Info

The **IStream** interface supports reading and writing data to stream objects. Stream objects contain the data in a structured storage object, where storages provide the structure. Simple data can be written directly to a stream, but most frequently, streams are elements nested within a storage object. They are similar to standard files.

The **IStream** interface defines methods similar to the MS-DOS FAT file functions. For example, each stream object has its own access rights and a seek pointer. The main difference between a stream object and a DOS file is that streams are not opened using a file handle, but through an **IStream** interface pointer.

The methods in this interface present your object's data as a contiguous sequence of bytes that you can read or write. There are also methods for committing and reverting changes on streams open in transacted mode and methods for restricting access to a range of bytes in the stream.

Streams can remain open for long periods of time without consuming file system resources. The **IStream::Release** method is similar to a close function on a file. Once released, the stream object is no longer valid and cannot be used.

When to Implement

Implement **IStream** on a container or object application when you require functionality not provided by the OLE compound file implementation. The specification of **IStream** defines more functionality that the OLE implementation supports. In addition, if you are creating a stream object that is larger than the heap in your machine's memory and you are using a global memory handle, the compound file implementation calls **GlobalReAlloc** internally whenever it needs more memory, which can be extremely inefficient. In this case, the preferred solution is to implement an **IStream** that uses memory allocated by **VirtualAlloc** instead of **GlobalAlloc**. This can reserve a large chunk of virtual address space and then commit memory within that address space as required. No data copying occurs and memory is committed only as it is needed. For more information, refer to **IStream - Compound File Implementation**.

When to Use

Call the methods of the **IStream** interface from a container or application to read and write the data for an object. Since stream objects can be marshaled to other processes, applications can share the data in storage objects without having to use global memory.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IStream Methods	Description
Read	Reads a specified number of bytes from the stream object into memory starting at the current seek pointer.
Write	Writes a specified number from bytes into the stream object starting at the current seek pointer.
Seek	Changes the seek pointer to a new location relative to the beginning of

	the stream, the end of the stream, or the current seek pointer.
SetSize	Changes the size of the stream object.
CopyTo	Copies a specified number of bytes from the current seek pointer in the stream to the current seek pointer in another stream.
Commit	Ensures that any changes made to a stream object open in transacted mode are reflected in the parent storage object.
Revert	Discards all changes that have been made to a transacted stream since the last **IStream::Commit** call.
LockRegion	Restricts access to a specified range of bytes in the stream. Supporting this functionality is optional since some file systems do not provide it.
UnlockRegion	Removes the access restriction on a range of bytes previously restricted with **IStream::LockRegion**.
Stat	Retrieves the **STATSTG** structure for this stream.
Clone	Creates a new stream object that references the same bytes as the original stream but provides a separate seek pointer to those bytes.

IStream::Clone

Quick Info

Creates a new stream object with its own seek pointer that references the same bytes as the original stream.

```
HRESULT Clone(
  IStream ** ppstm          //Points to location for pointer to the new stream object
);
```

Parameter

ppstm

[out] When successful, points to the location of an **IStream** pointer to the new stream object. If an error occurs, this parameter is NULL.

Return Values

S_OK

The stream was successfully cloned.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_INSUFFICIENT_MEMORY

The stream was not cloned due to a lack of memory.

STG_E_INVALIDPOINTER

The *ppStm* pointer is not valid.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

This method creates a new stream object for accessing the same bytes but using a separate seek pointer. The new stream object sees the same data as the source stream object. Changes written to one object are immediately visible in the other. Range locking is shared between the stream objects.

The initial setting of the seek pointer in the cloned stream instance is the same as the current setting of the seek pointer in the original stream at the time of the clone operation.

See Also

IStream - Compound File Implementation, **IStream::CopyTo**

IStream::Commit Quick Info

Ensures that any changes made to a stream object open in transacted mode are reflected in the parent storage. If the stream object is open in direct mode, **IStream::Commit** has no effect other than flushing all memory buffers to the next level storage object. The OLE compound file implementation of streams does not support opening streams in transacted mode.

```
HRESULT Commit(
  DWORD grfCommitFlags          //Specifies how changes are committed
);
```

Parameter

grfCommitFlags

[in] Controls how the changes for the stream object are committed. See the **STGC** enumeration for a definition of these values.

Return Values

S_OK

Changes to the stream object were successfully committed to the parent level.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_MEDIUMFULL

The commit operation failed due to lack of space on the storage device.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

This method ensures that changes to a stream object opened in transacted mode are reflected in the parent storage. Changes that have been made to the stream since it was opened or last committed are reflected to the parent storage object. If the parent is opened in transacted mode, the parent may still revert at a later time rolling back the changes to this stream object. The compound file implementation does not support opening streams in transacted mode, so this method has very little effect other than to flush memory buffers. For more information, refer to **IStream - Compound File Implementation**.

If the stream is open in direct mode, this method ensures that any memory buffers have been flushed out to the underlying storage object. This is much like a flush in traditional file systems.

The **IStream::Commit** method is useful on a direct mode stream when the implementation of the **IStream** interface is a wrapper for underlying file system APIs. In this case, **IStream::Commit** would be connected to the file system's flush call.

See Also

IStream - Compound File Implementation, **IStorage::Commit**

IStream::CopyTo Quick Info

Copies a specified number of bytes from the current seek pointer in the stream to the current seek pointer in another stream.

```
HRESULT CopyTo(
  IStream * pstm,                 //Points to the destination stream
  ULARGE_INTEGER cb,              //Specifies the number of bytes to copy
  ULARGE_INTEGER * pcbRead,       //Pointer to the actual number of bytes read from the source
  ULARGE_INTEGER * pcbWritten     //Pointer to the actual number of bytes written to the destination
);
```

Parameters

pstm

[in] Points to the destination stream. The stream pointed to by *pstm* can be a new stream or a clone of the source stream.

cb

[in] Specifies the number of bytes to copy from the source stream.

pcbRead

[out] Pointer to the location where this method writes the actual number of bytes read from the source. You can set this pointer to NULL to indicate that you are not interested in this value. In this case, this method does not provide the actual number of bytes read.

pcbWritten

[out] Pointer to the location where this method writes the actual number of bytes written to the destination. You can set this pointer to NULL to indicate that you are not interested in this value. In this case, this method does not provide the actual number of bytes written.

Return Values

S_OK

The stream object was successfully copied.

E_PENDING

Asynchronous Storage only: Part or all of the data to be copied is currently unavailable. For more information see **ILockBytes** and Asynchronous Storage.

STG_E_INVALIDPOINTER

The value of one of the pointer parameters is not valid.

STG_E_MEDIUMFULL

The stream is not copied because there is no space left on the storage device.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

This method copies the specified bytes from one stream to another. The seek pointer in each stream instance is adjusted for the number of bytes read or written. This method is equivalent to reading *cb* bytes into memory using **IStream::Read** and then immediately writing them to the destination stream using **IStream::Write**, although **IStream::CopyTo** will be more efficient.

The destination stream can be a clone of the source stream created by calling the **IStream::Clone** method.

If **IStream::CopyTo** returns an error, you cannot assume that the seek pointers are valid for either the source or destination. Additionally, the values of *pcbRead* and *pcbWritten* are not meaningful even though they are returned.

If **IStream::CopyTo** returns successfully, the actual number of bytes read and written are the same.

To copy the remainder of the source from the current seek pointer, specify the maximum large integer value for the *cb* parameter. If the seek pointer is the beginning of the stream, this technique copies the entire stream.

See Also

IStream - Compound File Implementation, **IStream::Read**, **IStream::Write**, **IStream::Clone**

IStream::LockRegion Quick Info

Restricts access to a specified range of bytes in the stream. Supporting this functionality is optional since some file systems do not provide it.

```
HRESULT LockRegion(
  ULARGE_INTEGER libOffset,      //Specifies the byte offset for the beginning of the range
  ULARGE_INTEGER cb,             //Specifies the length of the range in bytes
  DWORD dwLockType               //Specifies the restriction on accessing the specified range
);
```

Parameters

libOffset

[in] Integer that specifies the byte offset for the beginning of the range.

cb

[in] Integer that specifies the length of the range, in bytes, to be restricted.

dwLockType

[in] Specifies the restrictions being requested on accessing the range.

Return Values

S_OK

The specified range of bytes was locked.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **ILockBytes** and Asynchronous Storage.

STG_E_INVALIDFUNCTION

Locking is not supported at all or the specific type of lock requested is not supported.

STG_E_LOCKVIOLATION

Requested lock is supported, but cannot be granted because of an existing lock.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

The byte range can extend past the current end of the stream. Locking beyond the end of a stream is useful as a method of communication between different instances of the stream without changing data that is actually part of the stream.

Three types of locking can be supported: locking to exclude other writers, locking to exclude other readers or writers, and locking that allows only one requestor to obtain a lock on the given range, which is usually an alias for one of the other two lock types. A given stream instance might support either of the first two

types, or both. The lock type is specified by *dwLockType*, using a value from the **LOCKTYPE** enumeration.

Any region locked with **IStream::LockRegion** must later be explicitly unlocked by calling **IStream::UnlockRegion** with exactly the same values for the *libOffset*, *cb*, and *dwLockType* parameters. The region must be unlocked before the stream is released. Two adjacent regions cannot be locked separately and then unlocked with a single unlock call.

Notes to Callers

Since the type of locking supported is optional and can vary in different implementations of **IStream**, you must provide code to deal with the STG_E_INVALIDFUNCTION error.

This method has no effect in the compound file implementation, because the implementation does not support range locking.

Notes to Implementers

Support for this method is optional for implementations of stream objects since it may not be supported by the underlying file system. The type of locking supported is also optional. The STG_E_INVALIDFUNCTION error is returned if the requested type of locking is not supported.

See Also

IStream - Compound File Implementation, **LOCKTYPE**, **IStream::UnlockRegion**

IStream::Read Quick Info

Reads a specified number of bytes from the stream object into memory starting at the current seek pointer.

```
HRESULT Read(
  void * pv,          //Pointer to buffer into which the stream is read
  ULONG cb,           //Specifies the number of bytes to read
  ULONG * pcbRead     //Pointer to location that contains actual number of bytes read
);
```

Parameters

pv

[in] Points to the buffer into which the stream is read. If an error occurs, this value is NULL.

cb

[in] Specifies the number of bytes of data to attempt to read from the stream object.

pcbRead

[out] Pointer to a location where this method writes the actual number of bytes read from the stream object. You can set this pointer to NULL to indicate that you are not interested in this value. In this case, this method does not provide the actual number of bytes read.

Return Values

S_OK

Data was successfully read from the stream object.

S_FALSE

The data could not be read from the stream object.

E_PENDING

Asynchronous Storage only: Part or all of the data to be read is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for reading this stream object.

STG_E_INVALIDPOINTER

One of the pointer values is invalid.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

This method reads bytes from this stream object into memory. The stream object must be opened in STGM_READ mode. This method adjusts the seek pointer by the actual number of bytes read.

The number of bytes actually read is returned in the *pcbRead* parameter.

Notes to Callers

The actual number of bytes read can be fewer than the number of bytes requested if an error occurs or if the end of the stream is reached during the read operation.

Some implementations might return an error if the end of the stream is reached during the read. You must be prepared to deal with the error return or S_OK return values on end of stream reads.

See Also

IStream - Compound File Implementation, **STGMOVE**, **IStorage::OpenStream**, **IStream::Write**

IStream::Revert Quick Info

Discards all changes that have been made to a transacted stream since the last **IStream::Commit** call. On streams open in direct mode and streams using the OLE compound file implementation of **IStream::Revert**, this method has no effect.

HRESULT Revert(*void***);**

Return Values

S_OK

The stream was successfully reverted to its previous version.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

Remarks

This method discards changes made to a transacted stream since the last commit operation.

See Also

IStream - Compound File Implementation, **IStream::Commit**

IStream::Seek Quick Info

Changes the seek pointer to a new location relative to the beginning of the stream, to the end of the stream, or to the current seek pointer.

```
HRESULT Seek(
  LARGE_INTEGER dlibMove,                 //Offset relative to dwOrigin
  DWORD dwOrigin,                         //Specifies the origin for the offset
  ULARGE_INTEGER * plibNewPosition        //Pointer to location containing new seek pointer
);
```

Parameters

dlibMove

[in] Displacement to be added to the location indicated by *dwOrigin*. If *dwOrigin* is STREAM_SEEK_SET, this is interpreted as an unsigned value rather than signed.

dwOrigin

[in] Specifies the origin for the displacement specified in *dlibMove*. The origin can be the beginning of the file, the current seek pointer, or the end of the file. See the **STREAM_SEEK** enumeration for the values.

plibNewPosition

[out] Pointer to the location where this method writes the value of the new seek pointer from the beginning of the stream. You can set this pointer to NULL to indicate that you are not interested in this value. In this case, this method does not provide the new seek pointer.

Return Values

S_OK

The seek pointer has been successfully adjusted.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_INVALIDPOINTER

The value of the *plibNewPosition* parameter is not valid.

STG_E_INVALIDFUNCTION

The value of the *dwOrigin* parameter is not valid.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

IStream::Seek changes the seek pointer so subsequent reads and writes can take place at a different location in the stream object. It is an error to seek before the beginning of the stream. It is not, however, an error to seek past the end of the stream. Seeking past the end of the stream is useful for subsequent writes, as the stream will at that time be extended to the seek position immediately before the write is done.

You can also use this method to obtain the current value of the seek pointer by calling this method with the *dwOrigin* parameter set to STREAM_SEEK_CUR and the *dlibMove* parameter set to 0 so the seek pointer is not changed. The current seek pointer is returned in the *plibNewPosition* parameter.

See Also

IStream - Compound File Implementation, **STREAM_SEEK**, **IStream::Read**, **IStream::Write**

IStream::SetSize Quick Info

Changes the size of the stream object.

```
HRESULT SetSize(
  ULARGE_INTEGER libNewSize          //Specifies the new size of the stream object
);
```

Parameter

libNewSize

[in] Specifies the new size of the stream as a number of bytes.

Return Values

S_OK

The size of the stream object was successfully changed.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_MEDIUMFULL

The stream size is not changed because there is no space left on the storage device.

STG_E_INVALIDFUNCTION

The value of the *libNewSize* parameter is not valid. Since streams cannot be greater than 2^{32} bytes in the OLE-provided implementation, the high DWORD of *libNewSize* must be 0. If it is nonzero, this parameter is not valid.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

IStream::SetSize changes the size of the stream object. Call this method to preallocate space for the stream. If the *libNewSize* parameters larger than the current stream size, the stream is extended to the indicated size by filling the intervening space with bytes of undefined value. This operation is similar to the **IStream::Write** method if the seek pointer is past the current end-of-stream.

If the *libNewSize* parameter is smaller than the current stream, then the stream is truncated to the indicated size.

The seek pointer is not affected by the change in stream size.

Calling **IStream::SetSize** can be an effective way of trying to obtain a large chunk of contiguous space.

See Also

IStream - Compound File Implementation, **IStream::Write**

IStream::Stat Quick Info

Retrieves the **STATSTG** structure for this stream.

```
HRESULT Stat(
  STATSTG * pstatstg,          //Location for STATSTG structure
  DWORD grfStatFlag            //Values taken from the STATFLAG enumeration
);
```

Parameters

pstatstg

[out] Points to a **STATSTG** structure where this method places information about this stream object. This pointer is NULL if an error occurs.

grfStatFlag

[in] Specifies that this method does not return some of the fields in the **STATSTG** structure, thus saving a memory allocation operation. Values are taken from the **STATFLAG** enumeration.

Return Value

S_OK

The **STATSTG** structure was successfully returned at the specified location.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for accessing statistics for this storage object.

STG_E_INSUFFICIENTMEMORY

The STATSTG structure was not returned due to a lack of memory.

STG_E_INVALIDFLAG

The value for the *grfStateFlag* parameter is not valid.

STG_E_INVALIDPOINTER

The *pStatStg* pointer is not valid.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

IStream::Stat retrieves a pointer to the **STATSTG** structure that contains information about this open stream. When this stream is within a structured storage and **IStorage::EnumElements** is called, it creates an enumerator object with the **IEnumSTATSTG** interface on it, which can be called to enumerate the storages and streams through the STATSTG structures associated with each of them.

See Also

IStream - Compound File Implementation, **STATFLAG**, **STATSTG**

IStream::UnlockRegion Quick Info

Removes the access restriction on a range of bytes previously restricted with **IStream::LockRegion**.

```
HRESULT UnlockRegion(
  ULARGE_INTEGER libOffset,     //Specifies the byte offset for the beginning of the range
  ULARGE_INTEGER cb,            //Specifies the length of the range in bytes
  DWORD dwLockType              //Specifies the access restriction previously placed on the range
);
```

Parameters

libOffset

[in] Specifies the byte offset for the beginning of the range.

cb

[in] Specifies, in bytes, the length of the range to be restricted.

dwLockType

[in] Specifies the access restrictions previously placed on the range.

Return Values

S_OK

The byte range was unlocked.

E_PENDING

Asynchronous Storage only: Part or all of the stream's data is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_INVALIDFUNCTION

Locking is not supported at all or the specific type of lock requested is not supported.

STG_E_LOCKVIOLATION

The requested unlock cannot be granted.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

Remarks

IStream::UnlockRegion unlocks a region previously locked with the **IStream::LockRegion** method. Locked regions must later be explicitly unlocked by calling **IStream::UnlockRegion** with exactly the same values for the *libOffset*, *cb*, and *dwLockType* parameters. The region must be unlocked before the stream is released. Two adjacent regions cannot be locked separately and then unlocked with a single unlock call.

See Also

IStream - Compound File Implementation, **LOCKTYPE**, **IStream::LockRegion**

IStream::Write Quick Info

Writes a specified number from bytes into the stream object starting at the current seek pointer.

```
HRESULT Write(
  void const* pv,          //Pointer to buffer from which stream is written
  ULONG cb,                //Specifies the number of bytes to write
  ULONG * pcbWritten       //Specifies the actual number of bytes written
);
```

Parameters

pv

[in] Points to the buffer from which the stream should be written.

cb

[in] The number of bytes of data to attempt to write into the stream.

pcbWritten

[out] Pointer to a location where this method writes the actual number of bytes written to the stream object. The caller can set this pointer to NULL, in which case, this method does not provide the actual number of bytes written.

Return Values

S_OK

The data was successfully written into the stream object.

E_PENDING

Asynchronous Storage only: Part or all of the data to be written is currently unavailable. For more information see **IFillLockBytes** and Asynchronous Storage.

STG_E_MEDIUMFULL

The write operation was not completed because there is no space left on the storage device.

STG_E_ACCESSDENIED

The caller does not have sufficient permissions for writing this stream object.

STG_E_CANTSAVE

Data cannot be written for reasons other than no access or space.

STG_E_INVALIDPOINTER

One of the pointer values is invalid.

STG_E_REVERTED

The object has been invalidated by a revert operation above it in the transaction tree.

STG_E_WRITEFAULT

The write operation was not completed due to a disk error.

Remarks

IStream::Write writes the specified data to a stream object. The seek pointer is adjusted for the number of bytes actually written. The number of bytes actually written is returned in the *pcbWrite* parameter. If the byte count is zero bytes, the write operation has no effect.

If the seek pointer is currently past the end of the stream and the byte count is non-zero, this method increases the size of the stream to the seek pointer and writes the specified bytes starting at the seek pointer. The fill bytes written to the stream are not initialized to any particular value. This is the same as the end-of-file behavior in the MS-DOS FAT file system.

With a zero byte count and a seek pointer past the end of the stream, this method does not create the fill bytes to increase the stream to the seek pointer. In this case, you must call the **IStream::SetSize** method to increase the size of the stream and write the fill bytes.

The *pcbWrite* parameter can have a value even if an error occurs.

In the OLE-provided implementation, stream objects are not sparse. Any fill bytes are eventually allocated on the disk and assigned to the stream.

See Also

IStream - Compound File Implementation

IStream - Compound File Implementation

The **IStream** interface supports reading and writing data to stream objects. Stream objects contain the data in a structured storage object, where storages provide the structure. Simple data can be written directly to a stream, but most frequently, streams are elements nested within a storage object. They are similar to standard files.

The specification of **IStream** defines more functionality that the OLE implementation supports. For example, the **IStream** interface defines streams up to 2_{64} bytes in length requiring a 64-bit seek pointer. However, the OLE implementation only supports streams up to 2_{32} bytes in length and read and write operations are always limited to 2_{32} bytes at a time. The OLE implementation also does not support stream transactioning or region locking.

When you want to create a simple stream based on global memory, you can get an **IStream** pointer by calling the API function **CreateStreamOnHGlobal**. To get an **IStream** pointer within a compound file object, call either **StgCreateDocfile** or **StgOpenStorage**. These functions retrieve an **IStorage** pointer, with which you can then call **CreateStream**/**OpenStream** for an **IStream** pointer. In either case, the same **IStream** implementation code is used.

When to Use

Call the methods of **IStream** to read and write data to a stream.

Since stream objects can be marshaled to other processes, applications can share the data in storage objects without having to use global memory. In the OLE compound file implementation of stream objects, the custom marshaling facilities in OLE create a remote version of the original object in the new process when the two processes have shared memory access. Thus, the remote version does not need to communicate with the original process to carry out its functions.

The remote version of the stream object shares the same seek pointer as the original stream. If you do not want to share the seek pointer, you should use the **IStream::Clone** method to provide a copy of the stream object for the remote process.

> **Note** If you are creating a stream object that is larger than the heap in your machine's memory and you are using an HGLOBAL, the stream object calls **GlobalRealloc** internally whenever it needs more memory. Because **GlobalRealloc** always copies data from the source to the destination, increasing a stream object from 20M to 25M, for example, consumes immense amounts of time. This is due to the size of the increments copied and is worsened if there is less than 45M of memory on the machine because of disk swapping.
>
> The preferred solution is to implement an **IStream** that uses memory allocated by **VirtualAlloc** instead of **GlobalAlloc**. This can reserve a large chunk of virtual address space and then commit memory within that address space as required. No data copying occurs and memory is committed only as it is needed.
>
> Another alternative is to call the **IStream::SetSize** method on the stream object to increase the memory allocation in advance. This is not, however, as efficient as using **VirtualAlloc** as described above.

Remarks

IStream::Read

> Reads a specified number of bytes from the stream object into memory starting at the current seek pointer. This implementation returns S_OK if the end of the stream was reached during the read.

(This is the same as the "end of file" behavior found in the MS-DOS FAT file system.

IStream::Write

Writes a specified number from bytes into the stream object starting at the current seek pointer. In this implementation, stream objects are not sparse. Any fill bytes are eventually allocated on the disk and assigned to the stream.

IStream::Seek

Changes the seek pointer to a new location relative to the beginning of the stream, to the end of the stream, or to the current seek pointer.

IStream::SetSize

Changes the size of the stream object. In this implementation, there is no guarantee that the space allocated will be contiguous

IStream::CopyTo

Copies a specified number of bytes from the current seek pointer in the stream to the current seek pointer in another stream.

IStream::Commit

The compound file implementation of IStream supports opening streams only in direct mode, not transacted mode. Therefore, the method has no effect when called other than to flush all memory buffers to the next storage level.

In this implementation, it does not matter if you commit changes to streams, you need only commit changes for storage objects.

IStream::Revert

This implementation does not support transacted streams, so a call to this method has no effect.

IStream::LockRegion

Range-locking is not supported by this implementation, so a call to this method has no effect.

IStream::UnlockRegion

Removes the access restriction on a range of bytes previously restricted with **IStream::LockRegion**.

IStream::Stat

Retrieves the **STATSTG** structure for this stream

IStream::Clone

Creates a new stream object with its own seek pointer that references the same bytes as the original stream.

See Also

IStream, **IStorage**, **CreateStreamOnHGlobal**, **StgCreateDocfile**, **StgOpenStorage**

IUnknown Quick Info

The **IUnknown** interface lets clients get pointers to other interfaces on a given object through the **QueryInterface** method, and manage the existence of the object through the **IUnknown::AddRef** and **IUnknown::Release** methods. All other COM interfaces are inherited, directly or indirectly, from **IUnknown**. Therefore, the three methods in **IUnknown** are the first entries in the VTable for every interface.

When to Implement

You must implement **IUnknown** as part of every interface. If you are using C++ multiple inheritance to implement multiple interfaces, the various interfaces can share one implementation of **IUnknown**. If you are using nested classes to implement multiple interfaces, you must implement **IUnknown** once for each interface you implement.

When to Use

Use **IUnknown** methods to switch between interfaces on an object, add references, and release objects.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IUnknown::AddRef Quick Info

The **IUnknown::AddRef** method increments the reference count for an interface on an object. It should be called for every new copy of a pointer to an interface on a given object.

ULONG AddRef(*void***);**

Return Value

Returns an integer from 1 to n, the value of the new reference count. This information is meant to be used for diagnostic/testing purposes only, because, in certain situations, the value may be unstable.

Remarks

Objects use a reference counting mechanism to ensure that the lifetime of the object includes the lifetime of references to it. You use **IUnknown::AddRef** to stabilize a copy of an interface pointer. It can also be called when the life of a cloned pointer must extend beyond the lifetime of the original pointer. The cloned pointer must be released by calling **IUnknown::Release**.

Objects must be able to maintain $(2^{(31)})$-1 outstanding pointer references. Therefore, the internal reference counter that **IUnknown::AddRef** maintains must be a 32-bit unsigned integer.

Notes to Callers

Call this function for every new copy of an interface pointer that you make. For example, if you are passing a copy of a pointer back from a function, you must call **IUnknown::AddRef** on that pointer. You must also call **IUnknown::AddRef** on a pointer before passing it as an in-out parameter to a function; the function will call **IUnknown::Release** before copying the out-value on top of it.

See Also

IUnknown::Release

IUnknown::QueryInterface Quick Info

Returns a pointer to a specified interface on an object to which a client currently holds an interface pointer. This function must call **IUnknown::AddRef** on the pointer it returns.

```
HRESULT QueryInterface(
  REFIID iid,              //Identifier of the requested interface
  void ** ppvObject        //Indirect pointer to the object
);
```

Parameters

iid

[in] Identifier of the interface being requested.

ppvObject

[out] Indirectly points to the interface specified in *iid*. If the object does not support the interface specified in *iid*, **ppvObject* is set to NULL.

Return Value

S_OK if the interface is supported, E_NOINTERFACE if not.

Remarks

The **QueryInterface** method gives a client access to other interfaces on an object.

For any one object, a specific query for the **IUnknown** interface on any of the object's interfaces must always return the same pointer value. This allows a client to determine whether two pointers point to the same component by calling **QueryInterface** on both and comparing the results. It is specifically not the case that queries for interfaces (even the same interface through the same pointer) must return the same pointer value.

There are four requirements for implementations of **QueryInterface** (In these cases, "must succeed" means "must succeed barring catastrophic failure."):

- The set of interfaces accessible on an object through **IUnknown::QueryInterface** must be static, not dynamic. This means that if a call to **QueryInterface** for a pointer to a specified interface succeeds the first time, it must succeed again, and if it fails the first time, it must fail on all subsequent queries.
- It must be symmetric – if a client holds a pointer to an interface on an object, and queries for that interface, the call must succeed.
- It must be reflexive – if a client holding a pointer to one interface queries successfully for another, a query through the obtained pointer for the first interface must succeed.
- It must be transitive – if a client holding a pointer to one interface queries successfully for a second, and through that pointer queries successfully for a third interface, a query for the first interface through the pointer for the third interface must succeed.

IUnknown::Release Quick Info

Decrements the reference count for the calling interface on a object. If the reference count on the object falls to 0, the object is freed from memory.

ULONG Release(*void***);**

Return Value

Returns the resulting value of the reference count, which is used for diagnostic/testing purposes only. If you need to know that resources have been freed, use an interface with higher-level semantics.

Remarks

If **IUnknown::AddRef** has been called on this object's interface *n* times and this is the *n+1th* call to **IUnknown::Release**, the implementation of **IUnknown::AddRef** must cause the interface pointer to free itself. When the released pointer is the only existing reference to an object (whether the object supports single or multiple interfaces), the implementation must free the object.

Note Aggregation of objects restricts the ability to recover interface pointers.

Notes to Callers

Call this function when you no longer need to use an interface pointer. If you are writing a function that takes an in-out parameter, call **IUnknown::Release** on the pointer you are passing in before copying the out-value on top of it.

See Also

IUnknown::AddRef

IViewObject Quick Info

The **IViewObject** interface enables an object to display itself directly without passing a data object to the caller. In addition, this interface can create and manage a connection with an advise sink so the caller can be notified of changes in the view object.

The caller can request specific representations and specific target devices. For example, a caller can ask for either an object's content or an iconic representation. Also, the caller can ask the object to compose a picture for a target device that is independent of the drawing device context. As a result, the picture can be composed for one target device and drawn on another device context. For example, to provide a print preview operation, you can compose the drawing for a printer target device but actually draw the representation on the display.

The **IViewObject** interface is similar to **IDataObject**; except that **IViewObject** places a representation of the data onto a device context while **IDataObject** places the representation onto a transfer medium.

Unlike most other interfaces, **IViewObject** cannot be marshaled to another process. This is because device contexts are only effective in the context of one process.

When to Implement

Object handlers and in-process servers that manage their own presentations implement **IViewObject** for use by compound document containers. OLE provides an **IViewObject** implementation for its default object handler's cache.

When to Use

You call **IViewObject** from a container application if you need to draw a contained object on a specific device context. For example, if you want to print the object to a printer, you call the **Draw** method in the **IViewObject** interface.

Methods in VTable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IViewObject Methods	Description
Draw	Draws a representation of the object onto a device context.
GetColorSet	Returns the logical palette the object uses for drawing.
Freeze	Freezes the drawn representation of an object so it will not change until a subsequent **Unfreeze**.
Unfreeze	Unfreezes the drawn representation of an object.
SetAdvise	Sets up a connection between the view object and an advise sink so that the advise sink can receive notifications of changes in the view object.
GetAdvise	Returns the information on the most

recent **SetAdvise**.

IViewObject::Draw Quick Info

Draws a representation of an object onto the specified device context.

```
HRESULT Draw(
  DWORD dwAspect,                    //Aspect to be drawn
  LONG lindex,                       //Part of the object of interest in the draw operation
  void * pvAspect,                   //Pointer to DVASPECTINFO structure or NULL
  DVTARGETDEVICE * ptd,              //Pointer to target device in a structure
  HDC hicTargetDev,                  //Information context for the target device
  HDC hdcDraw,                       //Device context on which to draw
  const LPRECTL lprcBounds,          //Pointer to the rectangle in which the object is drawn
  const LPRECTL lprcWBounds,         //Pointer to the window extent and window origin when drawing a metafile
  BOOL (*) (DWORD) pfnContinue,      //Pointer to the callback function for canceling or continuing the drawing
  DWORD dwContinue                   //Value to pass to the callback function
);
```

Parameters

dwAspect

[in] Specifies the aspect to be drawn, that is, how the object is to be represented. Representations include content, an icon, a thumbnail, or a printed document. Valid values are taken from the enumerations **DVASPECT** and **DVASPECT2**. Note that newer objects and containers that support optimized drawing interfaces support the **DVASPECT2** enumeration values. Older objects and containers that do not support optimized drawing interfaces may not support **DVASPECT2**. Windowless objects allow only DVASPECT_CONTENT, DVASPECT_OPAQUE, and DVASPECT_TRANSPARENT.

lindex

[in] Portion of the object that is of interest for the draw operation. Its interpretation varies depending on the value in the *dwAspect* parameter. See the **DVASPECT** enumeration for more information.

pvAspect

[in] Pointer to additional information in a **DVASPECTINFO** structure that enables drawing optimizations depending on the aspect specified. Note that newer objects and containers that support optimized drawing interfaces support this parameter as well. Older objects and containers that do not support optimized drawing interfaces always specify NULL for this parameter.

ptd

[in] Pointer to the **DVTARGETDEVICE** structure that describes the device for which the object is to be rendered. If NULL, the view should be rendered for the default target device (typically the display). A value other than NULL is interpreted in conjunction with *hicTargetDev* and *hdcDraw*. For example, if *hdcDraw* specifies a printer as the device context, the *ptd* parameter points to a structure describing that printer device. The data may actually be printed if *hicTargetDev* is a valid value or it may be displayed in print preview mode if *hicTargetDev* is NULL.

hicTargetDev

[in] Information context for the target device indicated by the *ptd* parameter from which the object can extract device metrics and test the device's capabilities. If *ptd* is NULL; the object should ignore the value in the *hicTargetDev* parameter.

hdcDraw

[in] Device context on which to draw. For a windowless object, the *hdcDraw* parameter should be in MM_TEXT mapping mode with its logical coordinates matching the client coordinates of the containing window. For a windowless object, the device context should be in the same state as the one normally passed by a WM_PAINT message.

lprcBounds

[in] Pointer to a **RECTL** structure specifying the rectangle on *hdcDraw* and in which the object should be drawn. This parameter controls the positioning and stretching of the object. This parameter should be NULL to draw a windowless in-place active object. In every other situation, NULL is not a legal value and should result in an E_INVALIDARG error code. If the container passes a non-NULL value to a windowless object, the object should render the requested aspect into the specified device context and rectangle. A container can request this from a windowless object to render a second, non-active view of the object or to print the object.

lprcWBounds

[in] If *hdcDraw* is a metafile device context, pointer to a **RECTL** structure specifying the bounding rectangle in the underlying metafile. The rectangle structure contains the window extent and window origin. These values are useful for drawing metafiles. The rectangle indicated by *lprcBounds* is nested inside this *lprcWBounds* rectangle; they are in the same coordinate space.

If *hdcDraw* is not a metafile device context; *lprcWBounds* will be NULL.

pfnContinue

[in] Pointer to a callback function that the view object should call periodically during a lengthy drawing operation to determine whether the operation should continue or be canceled. This function returns TRUE to continue drawing. It returns FALSE to stop the drawing in which case **IViewObject::Draw** returns DRAW_E_ABORT.

dwContinue

[in] Value to pass as a parameter to the function pointed to by the *pfnContinue* parameter. Typically, *dwContinue* is a pointer to an application-defined structure needed inside the callback function.

Return Values

This method supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The object was drawn successfully.

OLE_E_BLANK

No data to draw from.

DRAW_E_ABORT

Draw operation aborted.

VIEW_E_DRAW

Error in drawing.

DV_E_LINDEX

Invalid value for *lindex*; currently only -1 is supported.

DV_E_DVASPECT

Invalid value for *dwAspect*.

OLE_E_INVALIDRECT

Invalid rectangle.

Remarks

A container application issues a call to **IViewObject::Draw** to create a representation of a contained object. This method draws the specified piece (*lindex*) of the specified view (*dwAspect* and *pvAspect*) on the specified device context (*hdcDraw*). Formatting, fonts, and other rendering decisions are made on the basis of the target device specified by the *ptd* parameter.

There is a relationship between the *dwDrawAspect* value and the *lprcbounds* value. The *lprcbounds* value specifies the rectangle on *hdcDraw* into which the drawing is to be mapped. For DVASPECT_THUMBNAIL, DVASPECT_ICON, and DVASPECT_SMALLICON, the object draws whatever it wants to draw, and it maps it into the space given in the best way. Some objects might scale to fit while some might scale to fit but preserve the aspect ratio. In addition, some might scale so the drawing appears at full width, but the bottom is cropped. The container can suggest a size via **IOleObject::SetExtent**, but it has no control over the rendering size. In the case of DVASPECT_CONTENT, the **Draw** implementation should either use the extents given by **IOleObject::SetExtent** or use the bounding rectangle given in the *lprcBounds* parameter.

For newer objects that support optimized drawing techniques and for windowless objects, this method should be used as follows:

- New drawing aspects are supported in *dwAspect* as defined in **DVASPECT2**.
- The *pvAspect* parameter can be used to pass additional information allowing drawing optimizations through the **DVASPECTINFO** structure.
- The **Draw** method can be called to redraw a windowless in-place active object by setting the *lrpcBounds* parameter to NULL. In every other situation, NULL is an illegal value and should result in an E_INVALIDARG error code. A windowless object uses the rectangle passed by the activation verb or calls **IOleInPlaceSite::SetObjectRects** instead of using this parameter. If the container passes a non-NULL value to a windowless object, the object should render the requested aspect into the specified device context and rectangle. A container can request this from a windowless object to render a second, non-active view of the object or to print the object. See the **IOleInPlaceSiteWindowless** interface for more information on drawing windowless objects.
- For windowless objects, the *dwAspect* parameter only allows the DVASPECT_CONTENT, DVASPECT_OPAQUE, and DVASPECT_TRANSPARENT aspects.
- For a windowless object, the *hdcDraw* parameter should be in MM_TEXT mapping mode with its logical coordinates matching the client coordinates of the containing window. For a windowless object, the device context should be in the same state as the one normally passed by a WM_PAINT message.

To maintain compatibility with older objects and containers that do not support drawing optimizations, all objects, rectangular or not, are required to maintain an origin and a rectangular extent. This allows the container to still consider all its embedded objects as rectangles and to pass them appropriate rendering rectangles in **IViewObjectEx::Draw**.

An object's extent depends on the drawing aspect. For non-rectangular objects, the extent should be the size of a rectangle covering the entire aspect. By convention, the origin of an object is the top-left corner of the rectangle of the DVASPECT_CONTENT aspect. In other words, the origin always coincides with the top-left corner of the rectangle maintained by the object's site, even for a non-rectangular object.

Note to Callers

The value of *hicTargetDevice* is typically an information context for the target device. However, it may be a full device context instead.

Note to Implementers

If you are writing an object handler (such as the default handler) that implements **IViewObject::Draw** by playing a metafile, you have to treat **SetPaletteEntries** metafile records in a special way because of Windows' behavior. The Windows function **PlayMetaFile** sets these palette entries to the foreground. You must override this default by setting them to the background palette. Use **EnumMetaFile** to do this. Enhanced metafiles are always recorded with the background palette so there is no need to do it manually.

See Also

DVASPECT, **DVASPECT2**, **DVASPECTINFO**, **IOleInPlaceSiteWindowless**, **OleDraw**

IViewObject::Freeze Quick Info

Freezes a certain aspect of the object's presentation so that it does not change until the **IViewObject::Unfreeze** method is called. The most common use of this method is for banded printing.

```
HRESULT Freeze(
  DWORD dwAspect,      //How the object is to be represented
  LONG lindex,         //Part of the object of interest in the draw operation
  void * pvAspect,     //Always NULL
  DWORD * pdwFreeze    //Points to location containing an identifying key
);
```

Parameters

dwAspect

[in] Specifies how the object is to be represented. Representations include content, an icon, a thumbnail, or a printed document. Valid values are taken from the enumeration **DVASPECT**. See the **DVASPECT** enumeration for more information.

lindex

[in] Portion of the object that is of interest for the draw operation. Its interpretation varies with *dwAspect.* See the **DVASPECT** enumeration for more information.

pvAspect

[in] Pointer to additional information about the view of the object specified in *dwAspect*. Since none of the current aspects support additional information, *pvAspect* must always be NULL.

pdwFreeze

[out] Pointer to where an identifying DWORD key is returned. This unique key is later used to cancel the freeze by calling **IViewObject::Unfreeze**. This key is an index that the default cache uses to keep track of which object is frozen.

Return Values

S_OK

The presentation was successfully frozen.

VIEW_S_ALREADYFROZEN

Presentation has already been frozen. The value of *pdwFreeze* is the identifying key of the already frozen object.

OLE_E_BLANK

Presentation not in cache.

DV_E_LINDEX

Invalid value for *lindex*; currently; only -1 is supported.

DV_E_DVASPECT

Invalid value for *dwAspect*.

Remarks

The **IViewObject::Freeze** method causes the view object to freeze its drawn representation until a subsequent call to **IViewObject::Unfreeze** releases it. After calling **IViewObject::Freeze**, successive calls to **IViewObject::Draw** with the same parameters produce the same picture until **IViewObject::Unfreeze** is called.

IViewObject::Freeze is not part of the persistent state of the object and does not continue across unloads and reloads of the object.

The most common use of this method is for banded printing.

While in a frozen state, view notifications are not sent. Pending view notifications are deferred to the subsequent call to **IViewObject::Unfreeze**.

See Also

DVASPECT, **IViewObject::Unfreeze**

IViewObject::GetAdvise Quick Info

Retrieves the existing advisory connection on the object if there is one. This method simply returns the parameters used in the most recent call to the **IViewObject::SetAdvise** method.

```
HRESULT GetAdvise(
  DWORD * pdwAspect,      //Pointer to where dwAspect parameter from previous SetAdvise call is returned
  DWORD * padvf,          //Pointer to where advf parameter from previous SetAdvise call is returned
  IAdviseSink ** ppAdvSink //Indirect pointer to interface on an advise sink
);
```

Parameters

pdwAspect

[out] Pointer to where the *dwAspect* parameter from the previous **SetAdvise** call is returned. If this pointer is NULL, the caller does not permit this value to be returned.

padvf

[out] Pointer to where the *advf* parameter from the previous **SetAdvise** call is returned. If this pointer is NULL, the caller does not permit this value to be returned.

ppAdvSink

[out] Indirect pointer to an **IAdviseSink** interface on an advise sink. The connection to this advise sink must have been established with a previous **SetAdvise** call, which provides the *pAdvSink* parameter. If this pointer is NULL, there is no established advisory connection.

Return Values

This method supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The existing advisory connection was retrieved.

See Also

ADVF, **IAdviseSink**, **IViewObject::SetAdvise**

IViewObject::GetColorSet Quick Info

Returns the logical palette that the object will use for drawing in its **IViewObject::Draw** method with the corresponding parameters.

```
HRESULT GetColorSet(
  DWORD dwAspect,                 //How the object is to be represented
  LONG lindex,                    //Part of the object of interest in the draw operation
  void * pvAspect,                //Always NULL
  DVTARGETDEVICE * ptd,           //Pointer to target device in a structure
  HDC hicTargetDev,               //Information context for the target device
  LOGPALETTE ** ppColorSet        //Indirect pointer to a structure
);
```

Parameters

dwAspect

[in] Specifies how the object is to be represented. Representations include content, an icon, a thumbnail, or a printed document. Valid values are taken from the enumeration **DVASPECT**. See the **DVASPECT** enumeration for more information.

lindex

[in] Portion of the object that is of interest for the draw operation. Its interpretation varies with *dwAspect.* See the **DVASPECT** enumeration for more information.

pvAspect

[in] Pointer to additional information about the view of the object specified in *dwAspect*. Since none of the current aspects support additional information, *pvAspect* must always be NULL.

ptd

[in] Pointer to the **DVTARGETDEVICE** structure that describes the device for which the object is to be rendered. If NULL, the view should be rendered for the default target device (typically the display). A value other than NULL is interpreted in conjunction with *hicTargetDev* and *hdcDraw*. For example, if *hdcDraw* specifies a printer as the device context, *ptd* points to a structure describing that printer device. The data may actually be printed if *hicTargetDev* is a valid value or it may be displayed in print preview mode if *hicTargetDev* is NULL.

hicTargetDev

[in] Information context for the target device indicated by the *ptd* parameter from which the object can extract device metrics and test the device's capabilities. If *ptd* is NULL, the object should ignore the *hicTargetDev* parameter.

ppColorSet

[out] Indirect pointer to where a **LOGPALETTE** structure is returned. The **LOGPALETTE** structure contains the set of colors that would be used if **IViewObject::Draw** were called with the same parameters for *dwAspect*, *lindex*, *pvAspect*, *ptd*, and *hicTargetDev*. A NULL pointer to the **LOGPALETTE** structure means that the object does not use a palette.

Return Values

This method supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the

following:

S_OK

The set of colors was returned successfully.

S_FALSE

Set of colors is empty or the object will not give out the information.

OLE_E_BLANK

No presentation data for object.

DV_E_LINDEX

Invalid value for *lindex*; currently only -1 is supported.

DV_E_DVASPECT

Invalid value for *dwAspect*.

Remarks

The **IViewObject::GetColorSet** method recursively queries any nested objects and returns a color set that represents the union of all colors requested. The color set eventually percolates to the top-level container that owns the window frame. This container can call **IViewObject::GetColorSet** on each of its embedded objects to obtain all the colors needed to draw the embedded objects. The container can use the color sets obtained in conjunction with other colors it needs for itself to set the overall color palette.

The OLE-provided implementation of **IViewObject::GetColorSet** looks at the data it has on hand to draw the picture. If CF_DIB is the drawing format, the palette found in the bitmap is used. For a regular bitmap, no color information is returned. If the drawing format is a metafile, the object handler enumerates the metafile looking for a CreatePalette metafile record. If one is found, the handler uses it as the color set.

Note to Implementers

Object applications that rely on the default handler for drawing and that use metafiles for doing so should provide a SetPaletteEntries record when they generate their metafiles. If a SetPaletteEntries record is not found, the default object handler returns S_FALSE.

See Also

DVASPECT

IViewObject::SetAdvise Quick Info

Sets up a connection between the view object and an advise sink so that the advise sink can be notified about changes in the object's view.

```
HRESULT SetAdvise(
  DWORD dwAspect,          //View for which notification is being requested
  DWORD advf,              //Information about the advise sink
  IAdviseSink * pAdvSink   //Pointer to the advise sink that is to receive change notifications
);
```

Parameters

dwAspect

[in] View for which the advisory connection is being set up. Valid values are taken from the enumeration **DVASPECT**. See the **DVASPECT** enumeration for more information.

advf

[in] Contains a group of flags for controlling the advisory connection. Valid values are from the enumeration **ADVF**. However, only some of the possible **ADVF** values are relevant for this method. The following table briefly describes the relevant values. See the **ADVF** enumeration for a more detailed description.

ADVF Value	Description
ADVF_ONLYONCE	Causes the advisory connection to be destroyed after the first notification is sent.
ADVF_PRIMEFIRST	Causes an initial notification to be sent regardless of whether data has changed from its current state.

Note that the ADVF_ONLYONCE and ADVF_PRIMEFIRST can be combined to provide an asynchronous call to **IDataObject::GetData**.

pAdvSink

[out] Pointer to the **IAdviseSink** interface on the advisory sink that is to be informed of changes. A NULL value deletes any existing advisory connection.

Return Values

This method supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The advisory connection was successfully established.

OLE_E_ADVISENOTSUPPORTED

Advisory notifications are not supported.

DV_E_DVASPECT

Invalid value for *dwAspect*.

Remarks

A container application that is requesting a draw operation on a view object can also register with the **IViewObject::SetAdvise** method to be notified when the presentation of the view object changes. To find out about when an object's underlying data changes, you must call **IDataObject::DAdvise** separately.

To remove an existing advisory connection, call the **IViewObject::SetAdvise** method with *pAdvSink* set to NULL.

If the view object changes, a call is made to the appropriate advise sink through its **IAdviseSink::OnViewChange** method.

At any time, a given view object can support only one advisory connection. Therefore, when **IViewObject::SetAdvise** is called and the view object is already holding on to an advise sink pointer, OLE releases the existing pointer before the new one is registered.

See Also

ADVF, **IAdviseSink**, **IViewObject::GetAdvise**

IViewObject::Unfreeze Quick Info

Releases a previously frozen drawing. The most common use of this method is for banded printing.

```
HRESULT Unfreeze(
  DWORD dwFreeze      //Contains key that determines view object to unfreeze
);
```

Parameter

dwFreeze

[in] Contains a key previously returned from **IViewObject::Freeze** that determines which view object to unfreeze.

Return Values

S_OK

The drawing was unfrozen successfully.

OLE_E_NOCONNECTION

Error in the unfreezing process or the object is currently not frozen.

See Also

IViewObject::Freeze

IViewObject2 Quick Info

The **IViewObject2** interface is an extension to the **IViewObject** interface which returns the size of the drawing for a given view of an object. You can prevent the object from being run if it isn't already running by calling this method instead of **IOleObject::GetExtent**.

Like the **IViewObject** interface, **IViewObject2** cannot be marshaled to another process. This is because device contexts are only effective in the context of one process.

The OLE-provided default implementation provides the size of the object in the cache.

When to Implement

Object handlers and in-process servers that manage their own presentations implement **IViewObject2** for use by compound document containers.

When to Use

A container application or object handler calls the **GetExtent** method in the **IViewObject2** interface to get the object's size from its cache.

Methods in Vtable Order

IUnknown Methods	Description
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IViewObject Methods	Description
Draw	Draws a representation of the object onto a device context.
GetColorSet	Returns the logical palette the object uses for drawing.
Freeze	Freezes the drawn representation of an object so it will not change until a subsequent **Unfreeze**.
Unfreeze	Unfreezes the drawn representation of an object.
SetAdvise	Sets up a connection between the view object and an advise sink so that the advise sink can receive notifications of changes in the view object.
GetAdvise	Returns the information on the most recent **SetAdvise**.

IViewObject2 Method	Description
GetExtent	Returns the size of the view object from the cache.

IViewObject2::GetExtent Quick Info

Returns the size that the specified view object will be drawn on the specified target device.

```
HRESULT GetExtent(
  DWORD dwAspect,          //View object for which the size is being requested
  DWORD lindex,            //Part of the object to draw
  DVTARGETDEVICE ptd,      //Pointer to the target device in a structure
  LPSIZEL lpsizel          //Pointer to size of object
);
```

Parameters

dwAspect

[in] Requested view of the object whose size is of interest. Valid values are taken from the enumerations **DVASPECT** and from **DVASPECT2**. Note that newer objects and containers that support optimized drawing interfaces support the **DVASPECT2** enumeration values. Older objects and containers that do not support optimized drawing interfaces may not support **DVASPECT2**.

lindex

[in] Portion of the object that is of interest. Currently only -1 is valid.

ptd

[in] Pointer to the **DVTARGETDEVICE** structure defining the target device for which the object's size should be returned.

lpsizel

[out] Pointer to where the object's size is returned.

Return Values

This method supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The object's extent was successfully returned.

OLE_E_BLANK

An appropriate cache is not available.

Remarks

The OLE-provided implementation of **IViewObject2::GetExtent** searches the cache for the size of the view object.

The **GetExtent** method in the **IOleObject** interface provides some of the same information as **IViewObject2::GetExtent**.

Note This method must return the same size as DVASPECT_CONTENT for all the new aspects in **DVASPECT2**. **IOleObject::GetExtent** must do the same thing.

If one of the new aspects is requested in *dwAspect*, this method can either fail or return the same rectangle as for the DVASPECT_CONTENT aspect.

Note to Callers

To prevent the object from being run if it isn't already running, you can call **IViewObject2::GetExtent** rather than **IOleObject::GetExtent** to determine the size of the presentation to be drawn.

See Also

DVASPECT, **DVASPECT2**, **IOleObject::GetExtent**, **IViewObject**

IViewObjectEx Quick Info

The **IViewObjectEx** interface is an extension derived from **IViewObject2** to provide support for:

- enhanced, flicker-free drawing for non-rectangular objects and transparent objects
- hit testing for non-rectangular objects
- control sizing

Flicker free drawing

Containers can now choose between a variety of drawing algorithms, depending on their sophistication and the situation.

Flicker is created by redrawing the background before letting an object redraw its foreground as in the back to front drawing algorithm known as the Painter's Algorithm. There are essentially two ways to avoid flickering:

- Draw into an offscreen bitmap and then copy the resulting image to the screen in one chunk. This technique might require significant additional resources to store the offscreen image, depending on the size of the region to drawn, the resolution, and the number of colors.
- Draw front to back, instead of back to front, excluding each rectangular area from the clipping region as soon as its has been painted. One benefit of this technique is that each pixel is painted only once. Speed depends essentially on the performance of the clipping support.

Each technique has advantages and disadvantages, depending on the specific situation. There is no single algorithm that is most efficient in all situations. Depending on the situation and their sophistication, containers may choose to use one or another, or a mix of both. The **IViewObjectEx** interface provides methods to support both techniques or a mixture of the two. Simple containers can implement a simplistic back to front painting algorithm directly to the screen. The speed is likely to be high but so will flicker. If flicker is to be reduced to a minimum, painting to an off-screen device context is the solution of choice. If memory consumption is a problem, containers can use clipping to reduce the use of off screen bitmaps.

To draw as flicker-free as possible without using an offscreen bitmap, the container will have to paint in two passes. The first pass is done front to back. During that pass, each object draws regions of itself that are cheap enough to clip out efficiently and that it can entirely obscure. These regions are known as opaque. After each object is done, the container clips out the regions just painted to ensure that subsequent objects will not modify the bits on the screen.

During the second pass, which occurs back to front, each object draws its remaining parts - irregular, oblique or in general difficult to clip out, such as text on transparent background. Such parts are known as transparent. At this point, the container is responsible for clipping out any opaque, already painted regions in front of the object currently drawing. The less painting during this second pass, the less flicker on the screen.

Clipping during the second pass may be very inefficient, since the clipping region needs to be recreated for every object that has something to draw. This might be acceptable if not too many overlapping objects have irregular or transparent parts. An object can tell its container ahead of time whether it wants to be called during this second pass or not.

If the container provides an offscreen bitmap to paint into, then it can skip the first pass and ask every object to render itself entirely during the second pass. In certain cases, the container may also decide than flicker is not a problem and use that same technique while painting directly on screen. For example, flicker might be acceptable when painting a form for the first time, but not when repainting.

Note Although documented here two pass drawing is not currently utilized by any containers.

Hit testing for non-rectangular objects

The **IViewEx** interface supports hit detection for non-rectangular objects. Using the **QueryHitPoint** and **QueryHitRect** methods, the object can participate in the hit-test logic with the container.

Control Sizing

The **IViewEx** interface allows controls to provide sizing hints as the user resizes the control. The control can specify a minimum and maximum size and can specify the nearest good size to a size requested by the user.

When to Implement

Implement this interface on objects that need to support efficient flicker-free drawing, non-rectangular hit testing, or control sizing. This interface is derived from the **IViewObject2** interface which, in turn, is derived from **IViewObject**.

All **IViewObjectEx** methods described in this document that take or return a position assume that the location is expressed in HIMETRIC units relative to the origin of the object.

When to Use

Containers call the methods of this interface to draw objects in an efficient, flicker free manner, test whether points or rectangles are within the object, or to resize controls.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments reference count.
Release	Decrements reference count.

IViewObject Methods	**Description**
Draw	Draws a representation of the object onto a device context.
GetColorSet	Returns the logical palette the object uses for drawing.
Freeze	Freezes the drawn representation of an object so it will not change until a subsequent **Unfreeze**.
Unfreeze	Unfreezes the drawn representation of an object.
SetAdvise	Sets up a connection between the view object and an advise sink so that the advise sink can receive notifications of changes in the view object.
GetAdvise	Returns the information on the most recent **SetAdvise**.

IViewObject2 Method	**Description**

GetExtent	Returns the size that the specified view object will be drawn on the specified target device.

IViewObjectEx Methods	Description
GetRect	Returns a rectangle describing a requested drawing aspect.
GetViewStatus	Returns information about the opacity of the object, and what drawing aspects are supported.
QueryHitPoint	Indicates whether a point is within a given aspect of an object.
QueryHitRect	Indicates whether any point in a rectangle is within a given drawing aspect of an object.
GetNaturalExtent	Provides sizing hints from the container for the object to use as the user resizes it.

See Also

IViewObject2

IViewObjectEx::GetNaturalExtent Quick Info

Provides sizing hints from the container for the object to use as the user resizes it.

```
HRESULT GetNaturalExtent(
    DWORD dwAspect,                 //Requested drawing aspect
    LONG lindex,                    //Portion of object for draw operation
    DVTARGETDEVICE* ptd,            //Pointer to structure describing target device
    HDC hicTargetDev,               //Information context for ptd
    DVEXTENTINFO*                   //Structure specifying sizing data
pExtentInfo,
    LPSIZEL* pSizel,                //Pointer to sizing data returned by object
  );
```

Parameters

dwAspect

[in] Requested drawing aspect. It can be any of the values from the **DVASPECT** enumeration.

lindex

[in] Indicates the portion of the object that is of interest for the draw operation. Its interpretation varies depending on the value in the *dwAspect* parameter. See the **DVASPECT** enumeration for more information.

ptd

[in] Pointer to the target device structure that describes the device for which the object is to be rendered. If NULL, the view should be rendered for the default target device (typically the display). A value other than NULL is interpreted in conjunction with *hicTargetDev* and *hdcDraw*. For example, if *hdcDraw* specifies a printer as the device context, the *ptd* parameter points to a structure describing that printer device. The data may actually be printed if *hicTargetDev* is a valid value or it may be displayed in print preview mode if *hicTargetDev* is NULL.

hicTargetDev

[in] Specifies the information context for the target device indicated by the *ptd* parameter from which the object can extract device metrics and test the device's capabilities. If *ptd* is NULL; the object should ignore the value in the *hicTargetDev* parameter.

pExtentInfo

[in] Pointer to **DVEXTENTINFO** structure that specifies the sizing data.

pSizel

[out] Pointer to sizing data returned by the object. The returned sizing data is set to -1 for any dimension that was not adjusted. That is if *cx* is -1 then the width was not adjusted, if *cy* is -1 then the height was not adjusted. If E_FAIL is returned indicating no size was adjusted then psizel may be NULL.

Return Values

S_OK

The sizing hints were successfully returned.

E_FAIL

This method is not implemented for the specified *dwAspect*, or the size was not adjusted.

E_NOTIMPL

This method was not implemented.

Remarks

There are two general approaches to sizing a control. The first approach gives the control responsibility for sizing itself; the second approach gives the container responsibility for sizing the control. The first approach is called autosizing. There are two alternatives involved in the second approach: content sizing and integral sizing.

The **IViewObjectEx::GetNaturalExtent** method supports both content and integral sizing. In content sizing, the container passes the **DVEXTENTINFO** structure to the object into which the object returns a suggested size. In integral sizing, the container passes a preferred size to the object in **DVEXTENTINFO**, and the object actually adjusts its height. Integral sizing is used when the user rubberbands a new size in design mode.

Autosizing typically occurs with objects such as the Label control which resizes if the autosize property was enabled and the associated text changed. Autosizing is handled differently depending on the state of the object.

If the object is inactive, the following occurs:

1. The object calls **IOleClientSite::RequestNewObjectLayout**.
2. The container calls **IOleObject::GetExtent** and retrieves the new extents
3. The container calls **IOleObject::SetExtent** and adjusts the new extents.

If the object is active, the following occurs:

1. The object calls **IOleInPlaceSite::OnPosRectChange** to specify that it requires resizing.
2. The container calls **IOleInPlaceObject::SetObjectRects** and specifies the new size.

The values of the *dwAspect* parameter can be one of the following **DVASPECT** enumeration values:

DVASPECT_CONTENT

Provide a representation of the control so it can be displayed as an embedded object inside of a container. This value is typically specified for compound document objects. The presentation can be provided for the screen or printer.

DVASPECT_DOCPRINT

Provide a representation of the control on the screen as though it were printed to a printer using the Print command from the File menu. The described data may represent a sequence of pages..

DVASPECT_ICON

Provide an iconic representation of the control.

DVASPECT_THUMBNAIL

Provide a thumbnail representation of an object so it can be displayed in a browsing tool. The thumbnail is approximately a 120 by 120 pixel, 16-color (recommended) device-independent bitmap potentially wrapped in a metafile.

See Also

DVASPECT, DVEXTENTINFO, IOleClientSite::RequestNewObjectLayout, IOleInPlaceObject::SetObjectRects, IOleInPlaceSite::OnPosRectChange, IOleObject::GetExtent, IOleObject::SetExtent,

IViewObjectEx::GetRect Quick Info

Returns a rectangle describing a requested drawing aspect.

```
HRESULT GetRect(
  DWORD dwAspect,      //Requested drawing aspect
  LPRECTL pRect        //Pointer to the rectangle
);
```

Parameters

dwAspect

[in] Drawing aspect requested.

pRect

[out] Pointer to the rectangle describing the requested drawing aspect.

Return Values

S_OK

The rectangle was successfully returned.

DV_E_DVASPECT

The method does not support the specified aspect. Either the object does not support the aspect requested or the aspect is not rectangular.

Remarks

This method returns a rectangle describing the specified drawing aspect. The returned rectangle is in HIMETRIC units, relative to the object's origin. The rectangle returned depends on the drawing aspect as follows:

DVASPECT_CONTENT

Objects should return the bounding rectangle of the whole object. The top-left corner is at the object's origin and the size is equal to the extent returned by **IViewObject2::GetExtent**.

DVASPECT_OPAQUE

Objects with a rectangular opaque region should return that rectangle. Others should fail and return error code DV_E_DVASPECT.

If a rectangle is returned, it is guaranteed to be completely obscured by calling **IViewObject::Draw** for that aspect. The container should use that rectangle to clip out the object's opaque parts before drawing any object behind it during the back to front pass. If this method fails on an object with a non-rectangular opaque region, the container should draw the entire object in the back to front part using the DVASPECT_CONTENT aspect.

DVASPECT_TRANSPARENT

Objects should return the rectangle covering all transparent or irregular parts. If the object does not have any transparent or irregular parts, it may return DV_E_ASPECT. A container may use this rectangle to determine whether there are other objects overlapping the transparent parts of a given object.

IViewObjectEx::GetViewStatus Quick Info

Returns information about the opacity of the object, and what drawing aspects are supported.

```
HRESULT GetViewStatus(
  DWORD* pdwStatus,        //Pointer to the view status
);
```

Parameters

pdwStatus

[out] Pointer to the view status. This information is returned as a combination of the **VIEWSTATUS** enumeration values.

Return Values

S_OK

The view status was successfully returned. This method should never fail or return an error code.

Remarks

In order to optimize the drawing process, the container needs to be able to determine whether an object is opaque and whether it has a solid background. It is not necessary to redraw objects that are entirely covered by a completely opaque object. Other operations, such as scrolling for example, can also be highly optimized if an object is opaque and has a solid background.

The **IViewObjectEx::GetViewStatus** method returns whether the object is entirely opaque or not (VIEWSTATUS_OPAQUE bit) and whether its background is solid (VIEWSTATUS_SOLIDBKGND bit). This information may change in time. An object may be opaque at a given time and become totally or partially transparent later on, for example, because of a change of the BackStyle property. An object should notify its sites when it changes using **IAdviseSinkEx::OnViewStatusChange** so the sites can cache this information for high speed access.

Objects not supporting **IViewObjectEx** are considered to be always transparent.

The **IViewObjectEx::GetViewStatus** method also returns a combination of bits indicating which aspects are supported.

If a given drawing aspect is not supported, all **IViewObjectEx** methods taking a drawing aspect as an input parameter should fail and return E_INVALIDARG. The **IViewObjectEx::GetViewStatus** method allows the container to get back information about all drawing aspects in one quick call. Normally the set of supported drawing aspects should not change with time. However, if this was not the case, an object should notify its container using **IAdviseSinkEx::OnViewStatusChange**.

Which drawing aspects are supported is independent of whether the object is opaque, partially transparent, or totally transparent. In particular, a transparent object that does not support DVASPECT_TRANSPARENT should be drawn correctly during the back to front pass using DVASPECT_CONTENT. However, this is likely to result in more flicker.

See Also

IAdviseSinkEx::OnViewStatusChange, **VIEWSTATUS**

IViewObjectEx::QueryHitPoint Quick Info

Indicates whether a point is within a given aspect of an object.

```
HRESULT QueryHitPoint(
  DWORD dwAspect,          //Requested drawing aspect
  LPRECT pRectBounds,      //Object bounding rectangle
  POINTL ptlLoc,           //Hit location
  LONG lCloseHint,         //Suggested distance considered close
  DWORD* pHitResult        //Pointer to returned hit information
);
```

Parameters

dwAspect

[in] Requested drawing aspect.

pRectBounds

[in] Object bounding rectangle in client coordinates of the containing window. This rectangle is computed and passed by the container so that the object can meaningfully interpret the hit location.

ptlLoc

[in] Hit location in client coordinates of the containing window.

lCloseHint

[in] Suggested distance in HIMETRIC units that the container considers close. This value is a hint, and objects can interpret it in their own way. Objects can also use this hint to roughly infer output resolution to choose expansiveness of hit test implementation.

pHitResult

[out] Pointer to returned information about the hit expressed as the **HITRESULT** enumeration values.

Return Values

S_OK

The hit information was successfully returned in *pHitResult*.

E_FAIL

This method is not implemented for the requested aspect. Use DVASPECT_CONTENT instead.

Remarks

To support hit detection on non-rectangular objects, the container needs a reliable way to ask an object whether or not a given location is inside one of its drawing aspects. This function is provided by **IViewObjectEx::QueryHitPoint**.

Note that since this method is part of the **IViewObjectEx** interface, the container can figure whether an mouse hit is over an object without having to necessarily launch the server. If the hit happens to be inside the object, then it is likely that the object will be in-place activated and the server started.

Typically, the container first quickly determines whether a given location is within the rectangular extent of an object. If the location is within the rectangular extent of an object, the container calls **IViewObjectEx::QueryHitPoint** to get confirmation that the location is actually inside the object. The hit location is passed in client coordinates of the container window. Since the object may be inactive when this method is called, the bounding rectangle of the object in the same coordinate system is also passed to this method, similarly to what happens in **IPointerInactive::OnInactiveSetCursor**.

Possible returned values include:

- outside, on a transparent region
- close enough to be considered a hit (may be used by small or thin objects)
- hit

QueryHitPoint is not concerned by the sub-objects of the object it is called for. It merely indicates whether the mouse hit was within the object or not.

QueryHitPoint can be called for any of the drawing aspects an object supports. It should fail if the it is not supported for the requested drawing aspect.

Transparent objects may wish to implement a complex hit-detection mechanism where the user can select either the transparent object or an object behind it, depending on where exactly the click happens inside the object. For example, a transparent TextBox showing big enough text may let the user select the object behind, for example, a bitmap, when the user clicks between the characters. For this reason, the information returned by **QueryHitPoint** includes indication about whether the hit happens on an opaque or transparent region.

An example of non-rectangular and transparent hit detection is a transparent circle control with an object behind it (a line in the example below):

{ewc msdncd, EWGraphic, bsd23522 0 /a "SDK.WMF"}

The values shown are for hit tests against the circle; gray regions are not part of the control, but are shown here to indicate an area around the image considered close. Each object implements its own definition of close but is assisted by a hint provided by the container so that closeness can be adjusted as images zoom larger or smaller.

In the picture above, the points marked "Hit", "Close" and "Transparent" would all be hits of varying strength on the circle, with the exception of the one marked "Transparent, (but for the line, close)." This illustrates the effect of the different strength of hits. Since the circle responds "transparent" while the line claims "close," and transparent is weaker than close, the line takes the hit.

Note to Implementers

An object supporting **IViewObjectEx** is required to implement this method at least for the DVASPECT_CONTENT aspect. The object should not take any other action in response to this method other than to return the information; there should be no side-effects.

See Also

IPointerInactive::OnInactiveSetCursor, **HITRESULT**

IViewObjectEx::QueryHitRect Quick Info

Indicates whether any point in a rectangle is within a given drawing aspect of an object.

```
HRESULT QueryHitRect(
  DWORD dwAspect,          //Requested drawing aspect
  LPRECT pRectBounds,      //Object bounding rectangle
  LPRECT pRectLoc,         //Hit location
  LONG lCloseHint,         //Suggested distance considered close
  DWORD* pHitResult        //Pointer to returned hit information
);
```

Parameters

dwAspect

[in] Requested drawing aspect.

pRectBounds

[in] Object bounding rectangle in client coordinates of the containing window. This rectangle is computed and passed by the container so that the object can meaningfully interpret the hit location.

pRectLoc

[in] Hit test rectangle in specified in HIMETRIC units, relative to the top-left corner of the object.

lCloseHint

[in] Suggested distance in HIMETRIC units that the container considers close. This value is a hint, and objects can interpret it in their own way. Objects can also use this hint to roughly infer output resolution to choose expansiveness of hit test implementation.

pHitResult

[out] Pointer to returned information about the hit expressed as the **HITRESULT** enumeration values.

Return Values

S_OK

The hit information was successfully returned in *pHitInfo*.

E_FAIL

This method is not implemented for the requested aspect. Use DVASPECT_CONTENT instead.

Remarks

Containers may need to test whether an object overlaps a given drawing aspect of another object. They can determine whether the objects overlap by requesting a region or at least a bounding rectangle of the aspect in question. However, a quicker way to do this is to call **IViewObjectEx::QueryHitRect** to ask the object whether a given rectangle intersects one of its drawing aspects.

Note Unlike **IViewObjectEx::QueryHitPoint**, this method does not return HITRESULT_TRANSPARENT or HITRESULT_CLOSE. It is strictly hit or miss, returning HITRESULT_OUTSIDE if no point in the rectangle is hit and HITRESULT_HIT if at least one point in

the rectangle is a hit.

Note to Implementers

An object supporting **IViewObjectEx** is required to implement this method at least for the DVASPECT_CONTENT aspect. The object should not take any other action in response to this method other than to return the information; there should be no side-effects. If there is any ambiguity about whether a point is a hit, for instance due to coordinates not converting exactly, the object should return HITRESULT_HIT whenever any point in the rectangle might be a hit on the object. That is, it is permissible to claim a hit for a point that is not actually rendered, but never correct to claim a miss for any point that is in the rendered image of the object.

See Also

HITRESULT

BindMoniker Quick Info

Locates an object by means of its moniker, activates the object if it is inactive, and retrieves a pointer to the specified interface on that object.

```
HRESULT BindMoniker(
  LPMONIKER pmk,          //Pointer to the object's moniker
  DWORD grfOpt,           //Reserved
  REFIID iidResult,       //Interface identifier
  LPVOID FAR *ppvResult   //Indirect pointer to requested interface
);
```

Parameters

pmk

[in] Pointer to the object's moniker.

grfOpt

[in] Reserved for future use; must be zero.

iidResult

[in] Interface identifier to be used to communicate with the object.

ppvResult

[out] Indirect pointer to the requested interface. If an error occurs, *ppvResult* is NULL. If the call is successful, the caller is responsible for releasing the pointer with a call to the object's **IUnknown::Release**.

Return Values

S_OK

The object was located and activated, if necessary, and that a pointer to the requested interface was returned.

MK_E_NOOBJECT

The object that the moniker object identified could not be found.

This function can also return any of the error values returned by the **IMoniker::BindToObject** method.

Remarks

BindMoniker is a helper function supplied as a convenient way for a client that has the moniker of an object to obtain a pointer to one of its interfaces. The **BindMoniker** function packages the following calls:

```
CreateBindCtx(0, &pbc);
pmk->BindToObject(pbc, NULL, riid, ppvObj);
```

CreateBindCtx creates a bind context object that supports the system implementation of **IBindContext**. The *pmk* parameter is actually a pointer to the **IMoniker** implementation on a moniker object. This implementation's **BindToObject** method supplies the pointer to the requested interface pointer.

If you have several monikers to bind in quick succession, and if you know that those monikers will activate the same object, it may be more efficient to call the **IMoniker::BindToObject** method directly, which allows you to use the same bind context object for all the monikers. See the **IBindCtx** interface for more information.

Container applications that allow their documents to contain linked objects are a special client that generally does not make direct calls to **IMoniker** methods. Instead, the client manipulates the linked objects through the **IOleLink** interface. The default handler implements this interface and calls the appropriate **IMoniker** methods as needed.

See Also

CreateBindCtx, **IMoniker::BindToObject**

CLSIDFromProgID Quick Info

Looks up a CLSID in the registry, given a ProgID.

```
HRESULT CLSIDFromProgID(
  LPCOLESTR lpszProgID,     //Pointer to the ProgID
  LPCLSID pclsid            //Pointer to the CLSID
);
```

Parameters

lpszProgID

[in] Pointer to the ProgID whose CLSID is requested.

pclsid

[out] Pointer to the retrieved CLSID on return.

Return Values

S_OK

The CLSID was retrieved successfully.

CO_E_CLASSSTRING

The registered CLSID for the ProgID is invalid.

REGDB_E_WRITEREGDB

An error occurred writing the CLSID to the registry. See "Remarks" below.

Remarks

Given a ProgID, **CLSIDFromProgID** looks up its associated CLSID in the registry. If the ProgID cannot be found in the registry, **CLSIDFromProgID** creates an OLE 1 CLSID for the ProgID and a CLSID entry in the registry. Because of the restrictions placed on OLE 1 CLSID values, **CLSIDFromProgID** and **CLSIDFromString** are the *only* two functions that can be used to generate a CLSID for an OLE 1 object.

See Also

ProgIDFromCLSID

CLSIDFromString

Quick Info

Converts a string generated by the **StringFromCLSID** function back into the original CLSID.

```
HRESULT CLSIDFromString(
  LPOLESTR lpsz,      //Pointer to the string representation of the CLSID
  LPCLSID pclsid      //Pointer to the CLSID
);
```

Parameters

lpsz

[in] Pointer to the string representation of the CLSID.

pclsid

[out] Pointer to the CLSID on return.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

NOERROR

The CLSID was obtained successfully.

CO_E_CLASSTRING

The class string was improperly formatted.

REGDB_E_WRITEREGDB

The CLSID corresponding to the class string was not found in the registry.

Remarks

Because of the restrictions placed on OLE 1 CLSID values, **CLSIDFromProgID** and **CLSIDFromString** are the *only* two functions that can be used to generate a CLSID for an OLE 1 object.

See Also

CLSIDFromProgID, **StringFromCLSID**

CoAddRefServerProcess

Increments a global per-process reference count.

ULONG CoAddRefServerProcess(void**);**

Return Values

S_OK

The CLSID was retrieved successfully.

Remarks

Servers can call **CoAddRefServerProcess** to increment a global per-process reference count. This function is particularly helpful to servers that are implemented with multiple threads, either multi-apartmented or free-threaded. Servers of these types must coordinate the decision to shut down with activation requests across multiple threads. Calling **CoAddRefServerProcess** increments a global per-process reference count, and calling **CoReleaseServerProcess** decrements that count.

When that count reaches zero, OLE automatically calls **CoSuspendClassObjects**, which prevents new activation requests from coming in. This permits the server to deregister its class objects from its various threads without worry that another activation request may come in. New activation requests result in launching a new instance of the local server process.

The simplest way for a local server application to make use of these API functions is to call **CoAddRefServerProcess** in the constructor for each of it's instance objects, and in each of its **IClassFactory::LockServer** methods when the *fLock* parameter is TRUE. The server application should also call **CoReleaseServerProcess** in the destruction of each of its instance objects, and in each of its **IClassFactory::LockServer** methods when the *fLock* parameter is FALSE. Finally, the server application should pay attention to the return code from **CoReleaseServerProcess** and if it returns 0, the server application should initiate its cleanup, which, for a server with multiple threads, typically means that it should signal its various threads to exit their message loops and call **CoRevokeClassObject** and **CoUninitialize**.

If these APIs are used at all, they must be called in both the object instances and the **LockServer** method, otherwise the server application may be shut down prematurely. In-process servers typically should not call **CoAddRefServerProcess** or **CoReleaseServerProcess**.

See Also

CoReleaseServerProcess, **IClassFactory::LockServer**, **Out-of-process Server Implementation Helpers**

CoBuildVersion Quick Info

This function is obsolete.

CoCopyProxy Quick Info

Makes a private copy of the specified proxy.

```
HRESULT CoCopyProxy(
  IUnknown *   punkProxy     //IUnknown pointer to the proxy to copy
  IUnknown **  ppunkCopy     //Indirect IUnknown pointer to the copy
);
```

Parameter

punkProxy

[in] Points to the **IUnknown** interface on the proxy to be copied. May not be NULL.

ppunkCopy

[out] Points to the location of the IUnknown pointer to the copy of the proxy. It may not be NULL.

Return Values

S_OK

Success.

E_INVALIDARG

One or more arguments are invalid.

Remarks

CoCopyProxy makes a private copy of the specified proxy. Typically, this is called when a client needs to change the authentication information of its proxy through a call to either **CoSetClientBlanket** or **IClientSecurity::SetBlanket** without changing this information for other clients. **CoSetClientBlanket** affects all the users of an instance of a proxy, so creating a private copy of the proxy through a call to **CoCopyProxy** eliminates the problem.

This function encapsulates the following sequence of common calls (error handling excluded):

```
pProxy->QueryInterface(IID_IClientSecurity, (void**)&pcs);
pcs->CopyProxy(punkProxy, ppunkCopy);
pcs->Release();
```

Local interfaces may not be copied. **IUnknown** and **IClientSecurity** are examples of existing local interfaces.

Copies of the same proxy have a special relationship with respect to **QueryInterface**. Given a proxy, *a*, of the *IA* interface of a remote object, suppose a copy of *a* is created, called *b*. In this case, calling QueryInterface from the *b* proxy for IID_IA will not retrieve the IA interface on *b*, but the one on *a*, the original proxy with the "default" security settings for the *IA* interface.

See Also

IClientSecurity::CopyProxy, **Security in COM**

CoCreateFreeThreadedMarshaler Quick Info

Creates an aggregatable object capable of context-dependent marshaling.

```
HRESULT CoCreateFreeThreadedMarshaler(
  LPUNKNOWN punkOuter,          // Pointer to object aggregating the marshaler object
  LPUNKNOWN * ppunkMarshaler    // Indirect pointer to the marshaler object
);
```

Parameters

punkOuter

[in] Pointer to the aggregating object's controlling **IUnknown**.

ppunkMarshaler

[out] Indirect pointer to the aggregatable marshaler's **IUnknown**.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The marshaler was created.

Remarks

The **CoCreateFreeThreadedMarshaler** function enables an object to efficiently marshal interface pointers between threads in the same process. If your objects do not support interthread marshaling, you have no need to call this function.

The **CoCreateFreeThreadedMarshaler** function performs the following tasks:

1. Creates a free-threaded marshaler object.
2. Aggregates this marshaler to the object specified by the *punkOuter* parameter. This object is normally the one whose interface pointers are to be marshaled.

The aggregating object's implementation of **IMarshal** should delegate **QueryInterface** calls for IID_IMarshal to the ***IUnknown*** of the free-threaded marshaler. Upon receiving a call, the free-threaded marshaler performs the following tasks:

1. Checks the destination context specified by the **CoMarshalInterface** function's *dwDestContext* parameter.
2. If the destination context is MSHCTX_INPROC, copies the interface pointer into the marshaling stream.
3. If the destination context is any other value, finds or creates an instance of COM's default (standard) marshaler and delegates marshaling to it.

Values for *dwDestContext* come from the MSHCTX enumeration. MSHCTX_INPROC indicates that the interface pointer is to be marshaled between different threads in the same process. Because both threads

have access to the same address space, the client thread can dereference the pointer directly rather than having to direct calls to a proxy. In all other cases, a proxy is required, so **CoCreateFreeThreadedMarshaler** delegates the marshaling job to COM's default implementation.

See Also

CoMarshalInterThreadInterfaceInStream, **CoGetInterfaceAndReleaseStream**

CoCreateGuid Quick Info

Creates a GUID, a unique 128-bit integer used for CLSIDs and interface identifiers.

```
HRESULT CoCreateGuid(
  GUID *pguid     //Pointer to the GUID on return
);
```

Parameter

pguid

[out] Pointer to the requested GUID on return.

Return Value

S_OK

The GUID was successfully created.

Win32 errors are returned by **UuidCreate** but wrapped as an HRESULT.

Remarks

The **CoCreateGuid** function calls the RPC function **UuidCreate**, which creates a GUID, a globally unique 128-bit integer. Use the **CoCreateGuid** function when you need an absolutely unique number that you will use as a persistent identifier in a distributed environment.To a very high degree of certainty, this function returns a unique value - no other invocation, on the same or any other system (networked or not), should return the same value.

See Also

UuidCreate (documented in the *RPC Programmer's Guide and Reference*)

CoCreateInstance Quick Info

Creates a single uninitialized object of the class associated with a specified CLSID. Call **CoCreateInstance** when you want to create only one object on the local system. To create a single object on a remote system, call **CoCreateInstanceEx**. To create multiple objects based on a single CLSID, refer to the **CoGetClassObject** function.

```
STDAPI CoCreateInstance(
  REFCLSID rclsid,          //Class identifier (CLSID) of the object
  LPUNKNOWN pUnkOuter,      //Pointer to whether object is or isn't part of an aggregate
  DWORD dwClsContext,       //Context for running executable code
  REFIID riid,              //Reference to the identifier of the interface
  LPVOID * ppv              //Indirect pointer to requested interface
);
```

Parameters

rclsid

[in] CLSID associated with the data and code that will be used to create the object.

pUnkOuter

[in] If **NULL**, indicates that the object is not being created as part of an aggregate. If non-**NULL**, pointer to the aggregate object's **IUnknown** interface (the controlling **IUnknown**).

dwClsContext

[in] Context in which the code that manages the newly created object will run. The values are taken from the enumeration **CLSCTX**.

riid

[in] Reference to the identifier of the interface to be used to communicate with the object.

ppv

[out] Indirect pointer to the requested interface.

Return Values

S_OK

An instance of the specified object class was successfully created.

REGDB_E_CLASSNOTREG

A specified class is not registered in the registration database. Also can indicate that the type of server you requested in the **CLSCTX** enumeration is not registered or the values for the server types in the registry are corrupt.

CLASS_E_NOAGGREGATION

This class cannot be created as part of an aggregate.

Remarks

The **CoCreateInstance** helper function provides a convenient shortcut by connecting to the class object

associated with the specified CLSID, creating an uninitialized instance, and releasing the class object. As such, it encapsulates the following functionality:

```
CoGetClassObject(rclsid, dwClsContext, NULL, IID_IClassFactory, &pCF);
hresult = pCF->CreateInstance(pUnkOuter, riid, ppvObj)
pCF->Release();
```

It is convenient to use **CoCreateInstance** when you need to create only a single instance of an object on the local machine. If you are creating an instance on remote machine, call **CoCreateInstanceEx**. When you are creating multiple instances, it is more efficient to obtain a pointer to the class object's **IClassFactory** interface and use its methods as needed. In the latter case, you should use the **CoGetClassObject** function.

In the **CLSCTX** enumeration, you can specify the type of server used to manage the object. The constants can be CLSCTX_INPROC_SERVER, CLSTCTX_INPROC_HANDLER, CLSCTX_LOCAL_SERVER, or any combination of these values. The constant CLSCTX_ALL is defined as the combination of all three. For more information about the use of one or a combination of these constants, refer to **CLSCTX**.

See Also

CoGetClassObject, **IClassFactory::CreateInstance**, **CoCreateInstanceEx**, **CLSCTX**, **Instance Creation Helper Functions**

CoCreateInstanceEx Quick Info

Creates an instance of a specific class on a specific machine.

```
HRESULT CoCreateInstanceEx(
  REFCLSID    rclsid,          //CLSID of the object to be created
  IUnknown *  punkOuter,       //If part of an aggregate, the controlling IUnknown
  DWORD       dwClsCtx,        //CLSCTX values
  COSERVERINFO* pServerInfo,   //Machine on which the object is to be instantiated
  ULONG       cmq,             //Number of MULTI_QI structures in rgmqResults
  MULTI_QI    rgmqResults      //Array of MULTI_QI structures
);
```

Parameters

rclsid

[in] CLSID of the object to be created.

punkOuter

[in] When non-NULL, indicates the instance is being created as part of an aggregate, and *punkOuter* is to be used as the new instance's controlling **IUnknown**. Aggregation is currently not supported cross-process or cross-machine. When instantiating an object out of process, CLASS_E_NOAGGREGATION will be returned if *punkOuter* is non-NULL.

dwClsCtx

[in] Values taken from the **CLSCTX** enumeration.

pServerInfo

[in] Machine on which to instantiate the object. May be NULL, in which case the object is instantiated on the current machine or at the machine specified in the registry under the class's **RemoteServerName** named-value, according to the interpretation of the *dwClsCtx* parameter. See the **CLSCTX** documentation for details).

cmq

[in] Number of MULTI_QI structures in *rgmqResults*. Must be greater than zero.

rgmqResults

Array of **MULTI_QI** structures. Each structure has three members: the identifier for a requested interface (*pIID*), the location to return the interface pointer (*pItf*) and the return value of the call to **QueryInterface** (*hr*).

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates success.

CO_S_NOTALLINTERFACES

At least one, but not all of the interfaces requested in the *rgmqResults* array were successfully

retrieved. The *hr* field of each of the MULTI_QI structures in *rgmqResults* indicates with S_OK or E_NOINTERFACE whether the specific interface was returned.

E_NOINTERFACE

None of the interfaces requested in the *rgmqResults* array were successfully retrieved.

Remarks

CoCreateInstanceEx creates a single uninitialized object associated with the given CLSID on a specified remote machine. This is an extension of the function **CoCreateInstance**, which creates an object on the local machine only. In addition, rather than requesting a single interface and obtaining a single pointer to that interface, **CoCreateInstanceEx** makes it possible to specify an array of structures, each pointing to an interface identifier (IID) on input, and, on return, containing (if available) a pointer to the requested interface and the return value of the **QueryInterface** call for that interface. This permits fewer round trips between machines.

The **CoCreateInstanceEx** helper function encapsulates three calls: first, to **CoGetClassObject** to connect to the class object associated with the specified CLSID, specifying the machine location of the class; second, to **IClassFactory::CreateInstance** to create an uninitialized instance, and finally, to **IClassFactory::Release**, to release the class object.

The object so created must still be initialized through a call to one of the initialization interfaces (such as **IPersistStorage:::Load**). The two helper functions, **CoGetInstanceFromFile** and **CoGetInstanceFromIStorage** encapsulate both the instance creation and initialization from the obvious sources.

See Also

CoGetInstanceFromFile, **CoGetInstanceFromIStorage**, **CLSCTX**, **COSERVERINFO**, **Instance Creation Helper Functions**

CoCreateStandardMalloc

This function is obsolete. Refer to **CoGetMalloc**.

CoDisconnectObject Quick Info

Disconnects all remote process connections being maintained on behalf of all the interface pointers that point to a specified object. Only the process that actually manages the object should call **CoDisconnectObject**.

```
STDAPI CoDisconnectObject(
  IUnknown * pUnk,        //Pointer to the interface on the object
  DWORD dwReserved        //Reserved for future use
);
```

Parameters

pUnk

[in] Pointer to any **IUnknown**-derived interface on the object to be disconnected.

dwReserved

[in] Reserved for future use; must be zero.

Return Values

S_OK

All connections to remote processes were successfully deleted.

Remarks

The **CoDisconnectObject** function enables a server to correctly disconnect all external clients to the object specified by *pUnk.*

The **CoDisconnectObject** function performs the following tasks:

1. Checks to see if the object to be disconnected implements the **IMarshal** interface. If so, it gets the pointer to that interface; if not, it gets a pointer to the standard marshaler's (*i.e.,* COM's) **IMarshal** implementation.
2. Using whichever **IMarshal** interface pointer it has acquired, the function then calls **IMarshal::DisconnectObject** to disconnect all out-of-process clients.

An object's client does not call **CoDisconnectObject** to disconnect itself from the server (clients should use **IUnknown::Release** for this purpose). Rather, an OLE server calls **CoDisconnectObject** to forcibly disconnect an object's clients, usually in response to a user closing the server application.

Similarly, an OLE container that supports external links to its embedded objects can call **CoDisconnectObject** to destroy those links. Again, this call is normally made in response to a user closing the application. The container should first call **IOleObject::Close** for all its OLE objects, each of which should send **IAdviseSink::OnClose** notifications to their various clients. Then the container can safely call **CoDisconnectObject** to close any existing connections.

See Also

IOleObject::Close, **IMarshal::DisconnectObject**

CoDosDateTimeToFileTime Quick Info

Converts the MS-DOS representation of the time and date to a **FILETIME** structure, which Win32 uses to determine the date and time.

```
BOOL CoDosDateTimeToFileTime(

  WORD nDosDate,          //16-bit MS-DOS date
  WORD nDosTime,          //16-bit MS-DOS time
  FILETIME * lpFileTime   //Pointer to the structure
);
```

Parameters

nDosDate

[in] 16-bit MS-DOS date.

nDosTime

[in] 16-bit MS-DOS time.

lpFileTime

[out] Pointer to the **FILETIME** structure.

Return Values

TRUE

The **FILETIME** structure was created successfully.

FALSE

The **FILETIME** structure was not created successfully, probably because of invalid arguments.

Remarks

The **FILETIME** structure and the **CoDosDateTimeToFileTime** and **CoFileTimeToDosDateTime** functions are part of the Win32 API definition. They are provided for compatibility in all OLE implementations, but are redundant on Win32 platforms.

MS-DOS records file dates and times as packed 16-bit values. An MS-DOS date has the following format:

Bits	Contents
0-4	Days of the month (1-31).
5-8	Months (1 = January, 2 = February, and so forth).
9-15	Year offset from 1980 (add 1980 to get actual year).

An MS-DOS time has the following format:

Bits	Contents
0-4	Seconds divided by 2.
5-10	Minutes (0-59).

11-15	Hours (0-23 on a 24-hour clock).

See Also

CoFileTimeToDosDateTime, CoFileTimeNow

CoFileTimeNow Quick Info

Returns the current time as a **FILETIME** structure.

```
HRESULT CoFileTimeNow(

  FILETIME  * lpFileTime     //Pointer to return the structure
);
```

Parameter

lpFileTime

[out] Pointer to return the **FILETIME** structure.

Return Values

S_OK

The current time was converted to a **FILETIME** structure.

See Also

CoDosDateTimeToFileTime, **CoFileTimeToDosDateTime**

CoFileTimeToDosDateTime Quick Info

Converts a **FILETIME** into MS-DOS date and time values.

```
BOOL CoFileTimeToDosDateTime(

  FILETIME   * lpFileTime,     //Pointer to the structure to be converted
  LPWORD lpDosDate,            //Pointer to the 16-bit MS-DOS date
  LPWORD lpDosTime             //Pointer to the 16-bit MS-DOS time
);
```

Parameters

lpFileTime

[in] Pointer to the **FILETIME** structure to be converted.

lpDosDate

[out] Pointer to the 16-bit MS-DOS date.

lpDosTime

[out] Pointer to the 16-bit MS-DOS time.

Return Values

TRUE

The **FILETIME** structure was converted successfully.

FALSE

The **FILETIME** structure was not converted successfully.

Remarks

This is the inverse of the operation provided by the **CoDosDateTimeToFileTime** function.

See Also

CoDosDateTimeToFileTime, **CoFileTimeNow**

CoFreeAllLibraries Quick Info

Frees all the DLLs that have been loaded with the **CoLoadLibrary** function (called internally by **CoGetClassObject**), regardless of whether they are currently in use. This function is usually not called directly, because **CoUninitialize** and **OleUninitialize** call it internally.

```
void CoFreeAllLibraries();
```

Remarks

To unload libraries, **CoFreeAllLibraries** uses a list of loaded DLLs for each process that the COM library maintains. The **CoUninitialize** function calls **CoFreeAllLibraries** internally, so OLE applications usually have no need to call this function directly.

See Also

CoLoadLibrary, **CoFreeLibrary**, **CoFreeUnusedLibraries**, **CoGetClassObject**, **CoUninitialize**, **OleUninitialize**

CoFreeLibrary Quick Info

Frees a library that, when loaded, was specified to be freed explicitly.

```
void CoFreeLibrary(

  HINSTANCE  hInst     //Handle of the library module to be freed
);
```

Parameter

hInst

[in] Handle to the library module to be freed, as returned by **CoLoadLibrary**.

Remarks

The **CoFreeLibrary** function should be called to free a library that is to be freed explicitly. This is established when the library is loaded with the *bAutoFree* parameter of **CoLoadLibrary** set to FALSE. It is illegal to free a library explicitly when the corresponding **CoLoadLibrary** call specifies that it be freed automatically (the *bAutoFree* parameter is set to TRUE).

See Also

CoFreeAllLibraries, **CoFreeUnusedLibraries**, **CoLoadLibrary**

CoFreeUnusedLibraries Quick Info

Unloads any DLLs that are no longer in use and that, when loaded, were specified to be freed automatically.

void CoFreeUnusedLibraries();

Remarks

Applications can call **CoFreeUnusedLibraries** periodically to free resources. It is most efficient to call it either at the top of a message loop or in some idle-time task. DLLs that are to be freed automatically have been loaded with the *bAutoFree* parameter of the **CoLoadLibrary** function set to TRUE. **CoFreeUnusedLibraries** internally calls **DllCanUnloadNow** for DLLs that implement and export that function.

See Also

CoFreeLibrary, **CoFreeUnusedLibraries**, **CoLoadLibrary**, **DLLCanUnloadNow**

CoGetCallContext Quick Info

Retrieves the context of the current call on the current thread.

```
HRESULT CoGetCallContext(

  REFIID   riid,     //Interface identifier
  void **  ppv      //Pointer to the requested interface
);
```

Parameters

riid

[in] Interface identifier (IID) of the call context that is being requested. If you are using the default call context supported by standard marshaling, only **IID_IServerSecurity** is available.

ppv

[out] Indirect pointer to the requested interface.

Return Values

S_OK

Success.

E_NOINTERFACE

The call context does not support the interface identified by *riid*.

Remarks

CoGetCallContext retrieves the context of the current call on the current thread. The *riid* parameter specifies the interface on the context to be retrieved. Currently, only **IServerSecurity** is available from the default call context supported by standard marshaling.

This is one of the functions provided to give the server access to any contextual information of the caller and to encapsulate common sequences of security checking and caller impersonation.

See Also

IServerSecurity, **Security in COM**

CoGetClassObject Quick Info

Provides a pointer to an interface on a class object associated with a specified CLSID. **CoGetClassObject** locates, and if necessary, dynamically loads the executable code required to do this.

Call **CoGetClassObject** directly when you want to create multiple objects through a class object for which there is a CLSID in the system registry. You can also retrieve a class object from a specific remote machine. Most class objects implement the **IClassFactory** interface. You would then call **IClassFactory::CreateInstance** to create an uninitialized object. It is not always necessary to go through this process. To create a single object, call instead the either the **CoCreateInstanceEx** function, which allows you to create an instance on a remote machine. This replaces the **CoCreateInstance** function, which can still be used to create an instance on a local machine. Both functions encapsulate connecting to the class object, creating the instance, and releasing the class object. Two other functions, **CoGetInstanceFromFile** and **CoGetInstanceFromIStorage**, provide both instance creation on a remote system, and object activation. OLE also provides many other ways to create an object in the form of numerous helper functions and interface methods whose function is to create objects of a single type and provide a pointer to an interface on that object.

```
STDAPI CoGetClassObject(

  REFCLSID rclsid,                    //CLSID associated with the class object
  DWORD dwClsContext,                 //Context for running executable code
  COSERVERINFO * pServerInfo,         //Pointer to machine on which the object is to be instantiated
  REFIID riid,                        //Reference to the identifier of the interface
  LPVOID * ppv                        //Indirect pointer to the interface
);
```

Parameters

rclsid

[in] CLSID associated with the data and code that you will use to create the objects.

dwClsContext

[in] Context in which the executable code is to be run. To enable a remote activation, CLSCTX_REMOTE_SERVER must be included. For more information on the context values and their use, see the **CLSCTX** enumeration.

pServerInfo

[in] Pointer to machine on which to instantiate the class object. May be NULL, in which case the class object is instantiated on the current machine or at the machine specified under the class's **RemoteServerName** key in the registry, according to the interpretation of the *dwClsCtx* parameter (see the CLSCTX documentation for details).

riid

[in] Reference to the identifier of the interface, which will be supplied in *ppv* on successful return. This interface will be used to communicate with the class object. Typically this value is **IID_IClassFactory**, although other values - such as **IID_IClassFactory2** which supports a form of licensing - are allowed. All OLE-defined interface IIDs are defined in the OLE header files as **IID_***interfacename*, where *interfacename* is the name of the interface.

ppv

[out] On successful return, indirect pointer to the requested interface.

Return Values

S_OK

Location and connection to the specified class object was successful.

REGDB_E_CLASSNOTREG

CLSID is not properly registered. Can also indicate that the value you specified in *dwClsContext* is not in the registry.

E_NOINTERFACE

Either the object pointed to by *ppv* does not support the interface identified by *riid*, or the **QueryInterface** operation on the class object returned E_NOINTERFACE.

REGDB_E_READREGDB

Error reading the registration database.

CO_E_DLLNOTFOUND

In-process DLL or handler DLL not found (depends on context).

CO_E_APPNOTFOUND

EXE not found (CLSCTX_LOCAL_SERVER only).

E_ACCESSDENIED

General access failure (returned from **LoadLib/CreateProcess**).

CO_E_ERRORINDLL

EXE has error in image.

CO_E_APPDIDNTREG

EXE was launched, but it didn't register class object (may or may not have shut down).

Remarks

A class object in OLE is an intermediate object that supports an interface that permits operations common to a group of objects. The objects in this group are instances derived from the same object definition represented by a single CLSID. Usually, the interface implemented on a class object is **IClassFactory**, through which you can create object instances of a given definition (class).

A call to **CoGetClassObject** creates, initializes, and gives the caller access (through a pointer to an interface specified with the *riid* parameter) to the class object. The class object is the one associated with the CLSID that you specify in the *rclsid* parameter. The details of how the system locates the associated code and data within a given machine are transparent to the caller, as is the dynamic loading of any code that is not already loaded.

If the class context is CLSCTX_REMOTE_SERVER, indicating remote activation is required, the **COSERVERINFO** structure provided in the *pServerInfo* parameter allows you to specify the machine on which the server is located. For information on the algorithm used to locate a remote server when *pServerInfo* is NULL, refer to the **CLSCTX** enumeration.

There are two places to find a CLSID for a given class:

- The registry holds an association between CLSIDs and file suffixes, and between CLSIDs and file signatures for determining the class of an object.
- When an object is saved to persistent storage, its CLSID is stored with its data.

To create and initialize embedded or linked OLE document objects, it is not necessary to call **CoGetClassObject** directly. Instead, call one of the **OleCreate** or **OleCreate***Xxx* helper functions. These functions encapsulate the entire object instantiation and initialization process, and call, among other functions, **CoGetClassObject**.

The *riid* parameter specifies the interface the client will use to communicate with the class object. In most cases, this interface is **IClassFactory**. This provides access to the **IClassFactory::CreateInstance** method, through which the caller can then create an uninitialized object of the kind specified in its implementation. All classes registered in the system with a CLSID must implement **IClassFactory**.

In rare cases, however, you may want to specify some other interface that defines operations common to a set of objects. For example, in the way OLE implements monikers, the interface on the class object is **IParseDisplayName**, used to transform the display name of an object into a moniker.

The *dwClsContext* parameter specifies the execution context, allowing one CLSID to be associated with different pieces of code in different execution contexts. The **CLSCTX** enumeration, defined in Compobj.H, specifies the available context flags. **CoGetClassObject** consults (as appropriate for the context indicated) both the registry and the class objects that are currently registered by calling the **CoRegisterClassObject** function.

To release a class object, use the class object's **Release** method. The function **CoRevokeClassObject** is to be used only to remove a class object's CLSID from the system registry.

See Also

CoCreateInstanceEx, **CoRegisterClassObject**, **CoRevokeClassObject**, **OleLoad**, **CLSCTX**, **Creating an Object through a Class Object**

CoGetCurrentProcess Quick Info

Returns a value that is unique to the current thread. It can be used to avoid PROCESSID reuse problems.

DWORD CoGetCurrentProcess();

Return Value

DWORD value

Unique value for the current thread that can be used to avoid PROCESSID reuse problems.

Remarks

The **CoGetCurrentProcess** function returns a value that is effectively unique, because it is not used again until $2^{(32)}$ more threads have been created on the current workstation or until the workstation is rebooted.

Using the value returned from a call to **CoGetCurrentProcess** can help you maintain tables that are keyed by threads or in uniquely identifying a thread to other threads or processes.

Using the value returned by **CoGetCurrentProcess** is more robust than using the HTASK task handle value returned by the Win32 function **GetCurrentTask**, because Windows task handles can be reused relatively quickly when a window's task dies.

CoGetInstanceFromFile Quick Info

Creates a new object and initializes it from a file using **IPersistFile::Load**.

```
HRESULT CoGetInstanceFromFile(
    COSERVERINFO *   pServerInfo,    //Pointer to COSERVERINFO struct indicating remote system
    CLSID*   pclsid,                 //Pointer to the class of the object to create
    IUnknown * punkOuter,            //If part of an aggregate, pointer to the controlling IUnknown
    DWORD dwClsCtx,                  //CLSCTX values
    OLECHAR* szName,                 //File to initialize the object with
    ULONG   cmq,                     //Number of MULTI_QI structures in rgmqResults
    MULTI_QI * rgmqResults           //Array of MULTI_QI structures
);
```

Parameters

pServerInfo

[in] Pointer to a **COSERVERINFO** structure that specifies the machine on which to instantiate the object and the authentication setting to be used. May be NULL, in which case the object is instantiated (1) on the current machine, (2) at the machine specified under the **RemoteServerName** named-value for the class in the registry, or (3) at the machine where the *szName* file resides if the **ActivateAtStorage** named-value is specified for the class in the registry or there is no local registry information.

pclsid

[in] Pointer to the class of the object to create. May be NULL, in which case there is a call to **GetClassFile**, using *szName* as its parameter to get the class of the object to be instantiated.

punkOuter

[in] When non-NULL, indicates the instance is being created as part of an aggregate, and *punkOuter* is to be used as the pointer to the new instance's controlling **IUnknown**. Aggregation is currently not supported cross-process or cross-machine. When instantiating an object out of process, CLASS_E_NOAGGREGATION will be returned if *punkOuter* is non-NULL.

dwClsCtx

[in] Values taken from the **CLSCTX** enumeration.

szName

[in] File to initialize the object with using **IPersistFile::Load**. May not be NULL.

cmq

[in] Number of **MULTI_QI** structures in *rgmqResults*. Must be greater than zero.

rgmqResults

[in] Array of MULTI_QI structures. Each structure has three members: the identifier for a requested interface (pIID), the location to return the interface pointer (*pItf*) and the return value of the call to **QueryInterface** (*hr*).

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates success.

CO_S_NOTALLINTERFACES

At least one, but not all of the interfaces requested in the *rgmqResults* array were successfully retrieved. The *hr* field of each of the MULTI_QI structures in *rgmqResults* indicates with S_OK or E_NOINTERFACE whether or not the specific interface was returned.

E_NOINTERFACE

None of the interfaces requested in the *rgmqResults* array were successfully retrieved.

Remarks

CoGetInstanceFromFile creates a new object and initializes it from a file using **IPersistFile::Load**. The result of this function is similar to creating an instance with a call to **CoCreateInstanceEx**, followed by an initializing call to **IPersistFile::Load**, with the following important distinctions:

- Fewer network round trips are required by this function when instantiating an object on a remote machine.
- In the case where *dwClsCtx* is set to CLSCTX_REMOTE_SERVER and *pServerInfo* is NULL, if the class is registered with the **ActivateAtStorage** sub-key or has no associated registry information, this function will instantiate an object on the machine where *szName* resides, providing the least possible network traffic. For example, if *szName* specified "\\myserver\users\johndo\file", the object would be instantiated on the "myserver" machine, and the object would access the file directly.

See Also

CoCreateInstanceEx, **CoGetInstanceFromIStorage**, **CLSCTX**, **Instance Creation Helper Functions**

CoGetInstanceFromIStorage Quick Info

Creates a new object and initializes it from a storage object through an internal call to **IPersistStorage::Load**.

```
HRESULT CoGetInstanceFromIStorage(
  COSERVERINFO *  pServerInfo,    //Pointer to COSERVERINFO struct indicating remote system
  CLSID *  pclsid,                //Pointer to the CLSID of the object to be created
  IUnknown * punkOuter,           //If part of an aggregate, pointer to the controlling IUnknown
  DWORD  dwClsCtx,                //Values taken from the CLSCTX enumeration
  IStorage * pstg,                //Pointer to storage from which object is to be initialized
  ULONG  cmq,                     //Number of MULTI_QI structures in rgmqResults
  MULTI_QI * rgmqResults          //Array of MULTI_QI structures
);
```

Parameters

pServerInfo

[in] Pointer to a **COSERVERINFO** structure that specifies the machine on which to instantiate the object and the authentication setting to be used. May be NULL, in which case the object is either instantiated (1) on the current machine, (2) at the machine specified under the **RemoteServerName** named-value for the class in the registry, or (3) at the machine where the storage object pointed to by *pstg* is located if the class is registered with **ActivateAtStorage** specified or has no local registry information.

pclsid

[in] Pointer to the class identifier (CLSID) of the object to be created. May be NULL, in which case there is a call to **IStorage:Stat** to find the class of the object.

punkOuter

[in] When non-NULL, indicates the instance is being created as part of an aggregate, and *punkOuter* is to be used as the pointer to the new instance's controlling **IUnknown**. Aggregation is currently not supported cross-process or cross-machine. When instantiating an object out of process, CLASS_E_NOAGGREGATION will be returned if *punkOuter* is non-NULL.

dwClsCtx

Values taken from the **CLSCTX** enumeration.

pstg

Pointer to storage to initialize the object with using **IPersistStorage::Load**. May not be NULL.

cmq

Number of MULTI_QI structures in *rgmqResults*. Must be greater than zero.

rgmqResults

Array of **MULTI_QI** structures. Each structure has three members: the identifier for a requested interface (pIID), the location to return the interface pointer (*pItf*) and the return value of the call to **QueryInterface** (*hr*).

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates success.

CO_S_NOTALLINTERFACES

At least one, but not all of the interfaces requested in the *rgmqResults* array were successfully retrieved. The *hr* field of each of the MULTI_QI structures in *rgmqResults* indicates with S_OK or E_NOINTERFACE whether the specific interface pointer was retrieved.

E_NOINTERFACE

None of the interfaces requested in the *rgmqResults* array were successfully retrieved.

Remarks

CoGetInstanceFromIStorage creates a new object and initializes it from a storage object through a call to **IPersistStorage::Load**. This function is similar to creating an instance using **CoCreateInstanceEx** followed by a call to **IPersistStorage::Load**, with the following important distinctions:

- Fewer network round trips are required by thisfunction when instantiating remotely.
- In the case where *dwClsCtx* is set to CLSCTX_REMOTE_SERVER and *pServerInfo* is NULL, if the class is registered with the **ActivateAtStorage** named value or has no associated registry information, this function will instantiate an object on the same machine where the storage object pointed to by *pstg* resides, providing the least possible network traffic. For example, if *pstg* were obtained through a call to **StgCreateDocfile**, specifying "\\myserver\users\johndo\file", the object would be instantiated on the "myserver" machine, and the object would access the storage object directly.

See Also

CoCreateInstanceEx, **CoGetInstanceFromFile**, **CLSCTX**, **Instance Creation Helper Functions**

CoGetInterfaceAndReleaseStream Quick Info

Unmarshals a buffer containing an interface pointer and releases the stream when an interface pointer has been marshaled from another thread to the calling thread.

```
HRESULT CoGetInterfaceAndReleaseStream(
  LPSTREAM  pStm,     //Pointer to the stream from which the object is to be marshaled
  REFIID riid,        //Reference to the identifier of the interface
  LPVOID * ppv        //Indirect pointer to the interface
);
```

Parameters

pStm

[in] Pointer to the **IStream** interface on the stream to be unmarshaled.

riid

[in] Reference to the identifier of the interface requested from the unmarshaled object.

ppv

[out] Indirect pointer to the unmarshaled interface.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates the output interface was unmarshaled and the stream was released.

This function can also return any of the values returned by **CoUnmarshalInterface**.

Remarks

The **CoGetInterfaceAndReleaseStream** function performs the following tasks:

1. Calls **CoUnmarshalInterface** to unmarshal an interface pointer previously passed in a call to **CoMarshalInterThreadInterfaceInStream**.
2. Releases the stream pointer. Even if the unmarshaling fails, the stream is still released because there is no effective way to recover from a failure of this kind.

See Also

CoMarshalInterThreadInterfaceInStream, **CoUnmarshalInterface**

CoGetMalloc Quick Info

Retrieves a pointer to the default OLE task memory allocator (which supports the system implementation of the **IMalloc** interface) so applications can call its methods to manage memory.

```
HRESULT CoGetMalloc(

  DWORD dwMemContext,      //Indicates if memory is private or shared
  LPMALLOC * ppMalloc      //Indirect pointer to memory allocator
);
```

Parameters

dwMemContext

[in] Reserved; value must be 1.

ppMalloc

[out] Indirect pointer to an IMalloc interface on the memory allocator.

Return Values

This function supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

Indicates the allocator was retrieved successfully.

Remarks

The pointer to the **IMalloc** interface pointer received through the *ppMalloc* parameter cannot be used from a remote process—each process must have its own allocator.

See Also

IMalloc, **CoTaskMemAlloc**

CoGetMarshalSizeMax Quick Info

Returns an upper bound on the number of bytes needed to marshal the specified interface pointer to the specified object.

```
STDAPI CoGetMarshalSizeMax(
  ULONG *pulSize,          //Pointer to the upper-bound value
  REFIID riid,             //Reference to the identifier of the interface
  IUnknown * pUnk,         //Pointer to the interface to be marshaled
  DWORD dwDestContext,     //Destination process
  LPVOID pvDestContext,    //Reserved for future use
  DWORD mshlflags          //Reason for marshaling
);
```

Parameters

pulSize

[out] Pointer to the upper-bound value on the size, in bytes, of the data packet to be written to the marshaling stream; a value of zero means that the size of the packet is unknown.

riid

[in] Reference to the identifier of the interface whose pointer is to be marshaled. This interface must be derived from the **IUnknown** interface.

pUnk

[in] Pointer to the interface to be marshaled; can be NULL. This interface must be derived from the **IUnknown** interface.

dwDestContext

[in] Destination context where the specified interface is to be unmarshaled. Values for *dwDestContext* come from the enumeration **MSHCTX**. Currently, unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext

[in] Reserved for future use; must be NULL.

mshlflags

[in] Flag indicating whether the data to be marshaled is to be transmitted back to the client process—the normal case—or written to a global table, where it can be retrieved by multiple clients. Values come from the enumeration **MSHLFLAGS**.

Return Values

This function supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

The upper bound was returned successfully.

CO_E_NOTINITIALIZED

The **CoInitialize** or **OleInitialize** function was not called on the current thread before this function was called.

Remarks

This function performs the following tasks:

1. Queries the object for an **IMarshal** pointer or, if the object does not implement **IMarshal**, gets a pointer to COM's standard marshaler.
2. Using whichever pointer is obtained in the preceding step, calls **IMarshal::GetMarshalSizeMax**.
3. Adds to the value returned by the call to **GetMarshalSizeMax** the size of the marshaling data header and, possibly, that of the proxy CLSID to obtain the maximum size in bytes of the amount of data to be written to the marshaling stream.

You do not explicitly call this function unless you are implementing **IMarshal**, in which case your marshaling stub should call this function to get the correct size of the data packet to be marshaled.

The value returned by this method is guaranteed to be valid only as long as the internal state of the object being marshaled does not change. Therefore, the actual marshaling should be done immediately after this function returns, or the stub runs the risk that the object, because of some change in state, might require more memory to marshal than it originally indicated.

See Also

CoMarshalInterface, **IMarshal::GetMarshalSizeMax**

CoGetPSClsid

This function returns the CLSID of the DLL that implements the proxy and stub for the specified interface.

```
WINOLEAPI CoGetPSClsid(
    REFIID riid,          // Interface whose proxy/stub CLSID is to be returned
    CLSID *pclsid         // Where to store returned proxy/stub CLSID
);
```

Parameters

riid

[in] The interface whose proxy/stub CLSID is to be returned.

pclsid

[out] Where to store the proxy/stub CLSID for the interface specified by *riid*.

Return Values

S_OK

The proxy/stub CLSID was successfully returned.

E_INVALIDARG

One of the parameters is invalid.

E_OUTOFMEMORY

There is insufficient memory to complete this operation.

Remarks

The **CoGetPSClsid** function looks at the HKEY_CLASSES_ROOT\Interfaces\{ string form of *riid* }\ ProxyStubClsid32 key in the registry to determine the CLSID of the DLL to load in order to create the proxy and stub for the interface specified by *riid*. This function also returns the CLSID for any interface IID registered by **CoRegisterPSClsid** within the current process.

See Also

CoRegisterPSClsid

CoGetStandardMarshal Quick Info

Creates a default, or standard, marshaling object in either the client process or the server process, depending on the caller, and returns a pointer to that object's **IMarshal** implementation.

```
STDAPI CoGetStandardMarshal(
  REFIID riid,                  //Reference to the identifier of the interface
  IUnknown * pUnk,              //Pointer to the interface to be marshaled
  DWORD dwDestContext,          //Destination process
  LPVOID pvDestContext,         //Reserved for future use
  DWORD mshlflags,              //Reason for marshaling
  LPMARSHAL * ppMarshal         //Indirect pointer to default IMarshal implementation
);
```

Parameters

riid

[in] Reference to the identifier of the interface whose pointer is to be marshaled. This interface must be derived from the **IUnknown** interface.

pUnk

[in] Pointer to the interface to be marshaled.

dwDestContext

[in] Destination context where the specified interface is to be unmarshaled. Values for *dwDestContext* come from the enumeration **MSHCTX**. Currently, unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext

[in] Reserved for future use; must be NULL.

mshlflags

[in] Flag indicating whether the data to be marshaled is to be transmitted back to the client process—the normal case—or written to a global table, where it can be retrieved by multiple clients. Valid values come from the **MSHLFLAGS** enumeration.

ppMarshal

[out] Indirect pointer to the standard marshaler.

Return Values

This function supports the standard return values E_FAIL, E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The **IMarshal** instance was returned successfully.

CO_E_NOTINITIALIZED

The **CoInitialize** or **OleInitialize** function was not called on the current thread before this function was

called.

Remarks

The **CoGetStandardMarshal** function creates a default, or standard, marshaling object in either the client process or the server process, as may be necessary, and returns that object's **IMarshal** pointer to the caller. If you implement **IMarshal**, you may want your implementation to call **CoGetStandardMarshal** as a way of delegating to COM's default implementation any destination contexts that you don't fully understand or want to handle. Otherwise, you can ignore this function, which COM calls as part of its internal marshaling procedures.

When the COM library in the client process receives a marshaled interface pointer, it looks for a CLSID to be used in creating a proxy for the purposes of unmarshaling the packet. If the packet does not contain a CLSID for the proxy, COM calls **CoGetStandardMarshal,** passing a NULL *pUnk* value. This function creates a standard proxy in the client process and returns a pointer to that proxy's implementation of **IMarshal**. COM uses this pointer to call **CoUnmarshalInterface** to retrieve the pointer to the requested interface.

If your OLE server application's implementation of **IMarshal** calls **CoGetStandardMarshal**, you should pass both the IID of (*riid*), and a pointer to (*pUnk*), the interface being requested.

This function performs the following tasks:

1. Determines whether *pUnk* is NULL.
2. If *pUnk* is NULL, creates a standard interface proxy in the client process for the specified *riid* and returns the proxy's **IMarshal** pointer.
3. If *pUnk* is not NULL, checks to see if a marshaler for the object already exists, creates a new one if necessary, and returns the marshaler's **IMarshal** pointer.

See Also

IMarshal

CoGetTreatAsClass Quick Info

Returns the CLSID of an object that can emulate the specified object.

```
HRESULT CoGetTreatAsClass(
    REFCLSID clsidOld,      //CLSID of object that is being emulated
    LPCLSID pclsidNew       //Pointer to CLSID for object that can emulate clsidOld
);
```

Parameters

clsidOld

[in] CLSID of the object that can be emulated (treated as) an object with a different CLSID.

pclsidNew

[out] Pointer to where the CLSID that can emulate *clsidOld* objects is retrieved. This parameter cannot be NULL. If there is no emulation information for *clsidOld* objects, the *clsidOld* parameter is supplied.

Return Values

S_OK

A new CLSID was successfully returned.

S_FALSE

No emulation information for the *clsidOld* parameter and that the *pclsidNew* parameter is set to *clsidOld*.

REGDB_E_READREGDB

An error reading the registry.

This function can also return any of the error values returned by the **CLSIDFromString** function.

Remarks

CoGetTreatAsClass returns the **TreatAs** entry in the registry for the specified object. The **TreatAs** entry, if set, is the CLSID of a registered object (an application) that can emulate the object in question. The **TreatAs** entry is set through a call to the **CoTreatAsClass** function. Emulation allows an application to open and edit an object of a different format, while retaining the original format of the object. Objects of the original CLSID are activated and treated as objects of the second CLSID. When the object is saved, this may result in loss of edits not supported by the original format. If there is no **TreatAs** entry for the specfied object, this function returns the CLSID of the original object (*clsidOld*).

See Also

CoTreatAsClass

CoImpersonateClient Quick Info

Allows the server to impersonate the client of the current call for the duration of the call.

HRESULT CoImpersonateClient()

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates success.

Remarks

Allows the server to impersonate the client of the current call for the duration of the call. If you do not call CoRevertToSelf, OLE reverts automatically for you. This function will fail unless the object is being called with RPC_C_AUTHN_LEVEL_CONNECT or higher authentication in effect (any authentication level except RPC_C_AUTHN_LEVEL_NONE) This function encapsulates the following sequence of common calls (error handling excluded):

```
CoGetCallContext(IID_IServerSecurity, (void**)&pss);
pss->ImpersonateClient();
pss->Release();
```

This helper function encapsulates the process of getting a pointer to an instance of **IServerSecurity** that contains data about the current call, calling its **ImpersonateClient** method, and then releasing the pointer.

See Also

IServerSecurity::ImpersonateClient, **Security in COM**

CoInitialize Quick Info

The **CoInitialize** function initializes the Component Object Model(COM) library. You must initialize the library before you can call its functions. Applications must call **CoInitialize** before they make any other COM library calls except the **CoGetMalloc** function and memory allocation calls.

```
HRESULT CoInitialize(
  LPVOID pvReserved      //Reserved, must be NULL
);
```

Parameter

pvReserved

[in] In 32-bit OLE, this parameter must be NULL. The 32-bit version of OLE does not support applications replacing OLE's allocator and if the parameter is not NULL, **CoInitialize** returns E_INVALIDARG.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The library was initialized successfully.

S_FALSE

The library is already initialized or that it could not release the default allocator.

Remarks

You need to call this before you call any of the OLE library functions (except **CoGetMalloc**, to get a pointer to the standard allocator, and the memory allocation functions and methods) unless you call the **OleInitialize** function, which calls **CoInitialize** internally.

Typically, **CoInitialize** is called only once in the process that uses the OLE library. There can be multiple calls, but subsequent calls return S_FALSE. To close the library gracefully, each successful call to **CoInitialize**, including those that return S_FALSE, *must* be balanced by a corresponding call to its companion helper function, **CoUninitialize**.

See Also

CoUninitialize, **OleInitialize**, **Processes and Threads**

CoInitializeEx Quick Info

Initializes the Component Object Model (COM) for use by the current thread. You can call **CoinitializeEx** in preference to calling **CoInitialize**, the implementation of which simply calls **CoInitializeEx**, specifying COINIT_APARTMENTTHREADED.

```
HRESULT CoInitializeEx(
  void *   pvReserved,    //Reserved
  DWORD    dwCoInit       //COINIT value
);
```

Parameters

pvReserved

[in] Reserved for future use; must be NULL.

dwCoInit

[in] This may contain any set of values from the COINIT enumeration except for both apartment and multi-threaded.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates success.

RPC_E_CHANGED_MODE

A previous call to **CoInitializeEx** specified a concurrency model for this thread different from the one currently specified for this thread.

Remarks

CoInitializeEx initializes the Component Object Model (COM) for use by the current thread. The *dwCoInit* parameter specifies the type of concurrency control - *multi-threaded* or *apartment-threaded* - required by objects created by this thread. A call to **CoInitializeEx** specifying COINIT_APARTMENTTHREADED is equivalent to a call to **CoInitialize**, because prior to NT 4.0, the default concurrency control required by objects was *apartment-threaded*.

Objects created on a *multi-threaded* COM thread must be able to receive calls on their methods from other threads at any time. Typically, you would implement some form of concurrency control in a multi-threaded object's code using Win32 synchronization primitives, such as *critical sections*, *semaphores*, or *mutexes*, to protect the object's data. Objects created on an *apartment-threaded* COM thread receive calls on their methods from their apartment's thread only, so calls are serialized, and calls only arrive at message-queue boundaries (**PeekMessage**, **SendMessage**).

Applications must call **CoInitializeEx** or **CoInitialize** before making any other COM library calls except the **CoGetMalloc** function and other memory allocation calls (**CoTaskMemAlloc**, **CoTaskMemFree**, **CoTaskMemReAlloc**, and the **IMalloc** methods on the task allocator supplied by **CoGetMalloc**).

Typically, **CoInitializeEx** is called only once by each thread in the process that uses the OLE library.

Multiple calls by the same thread are allowed so long as they pass the same concurrency flag, but subsequent valid calls return S_FALSE. To close the library gracefully, each successful call to **CoInitialize** or **CoInitializeEx**, including those that return E_INVALIDARG, *must* be balanced by a corresponding call to **CoUninitialize**.

Internally, the **OleInitialize** function calls **CoInitializeEx** with the COINIT_APARTMENTTHREADED flag. This implies that a thread that uses **CoInitializeEx** to initialize a thread for *multi-threaded* object concurrency may not use the features enabled by **OleInitialize**, because **OleInitialize** will fail.

See Also

COINIT, **CoInitialize**, **Processes and Threads**

CoInitializeSecurity Quick Info

Registers security and sets the default security values. For legacy applications, COM automatically calls this function with values from the registry.

```
HRESULT CoInitializeSecurity(
  PSECURITY_DESCRIPTOR     pVoid,                //Points to security descriptor
  DWORD    cAuthSvc,                             //Count of entries in asAuthSvc
  SOLE_AUTHENTICATION_SERVICE * asAuthSvc,       //Array of names to register
  void * pReserved1,                             //Reserved for future use
  DWORD    dwAuthnLevel,                         //The default authentication level for proxies
  DWORD    dwImpLevel,                           //The default impersonation level for proxies
  RPC_AUTH_IDENTITY_HANDLE    pAuthInfo,         //Reserved; must be set to NULL
  DWORD    dwCapabilities,                       //Additional client and/or server-side capabilities
  void * pvReserved2                             //Reserved for future use
);
```

Parameters

pVoid

[in] Security descriptor. If NULL, no ACL checking will be done. If not NULL, COM will check ACLs on new connections. If not NULL, *dwAuthnLevel* cannot be RPC_C_AUTHN_LEVEL_NONE.

cAuthSvc

[in] Count of entries in *asAuthSvc*. Zero means register no services. A value of -1 tells COM to choose which authentication services to register.

asAuthSvc

[in] Array of authentication/authorization/principal names to register. These values are registered to allow incoming calls. After that they are ignored. The default authentication/authorization/principal for each proxy will be negotiated regardless of whether these are set. For example, if the application registers RPC_C_AUTHN_WINNT and receives and interface from a machine that only supports RPC_C_AUTHN_DEC_PUBLIC, COM will choose RPC_C_AUTHN_DEC_PUBLIC if this machine supports it.

pReserved1

[in] Reserved for future use; must be NULL.

dwAuthnLevel

[in] The default authentication level for proxies. On the server side, COM will fail calls that arrive at a lower level. All calls to **AddRef** and **Release** are made at this level.

dwImpLevel

[in] The default impersonation level for proxies. This value is not checked on the server side. **AddRef** and **Release** calls are made with this impersonation level so even security aware apps should set this carefully. Setting **IUnknown** security only affects calls to **QueryInterface**, not **AddRef** or **Release**.

pAuthInfo

[in] Reserved for future use; must be NULL.

dwCapabilities

[in] Additional client and/or server-side capabilities. Any set of EOAC flags may be passed. Currently only EOAC_MUTUAL_AUTH, EOAC_SECURE_REFS, and EOAC_NONE are defined.

pReserved2

[in] Reserved for future use; must be zero.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates success.

Remarks

The **CoInitializeSecurity** layer initializes the security layer and sets the specified values as the security default. The *pSecDesc* parameter contains two ACLs. The discretionary ACL (DACL) indicates who is allowed to call this process and who is explicitly denied. The system ACL (SACL) contains audit information; this is not supported in the current release, so this portion of *pSecDesc* must be NULL, so there is no auditing.

A NULL DACL will allow calls from anyone. A DACL with no ACEs allows no access. For information on ACLs and ACEs, refer to Win32 Programmers Reference/Overviews/System Services/Security/Security/Security Model.

The owner and group of the SECURITY_DESCRIPTOR must be set – applications should call **AccessCheck** (not **IsValidSecurityDescriptor**) to ensure that their security descriptor is correctly formed prior to calling **CoInitializeSecurity**.

If the application passes a NULL security descriptor, COM will construct one that allows calls from the current user and local system. All new connections will be audited. Distributed COM will copy the security descriptor.

If mutual authentication is enabled all calls will fail unless the server identity is verified to match the principal name set on the proxy. Without mutual authentication, security only helps the server; the client has no idea who is handling his call. While **CoInitializeSecurity** takes principal names as parameters, that does not mean that the server can register any arbitrary name. The security provider verifies that the server has a right to use the names registered.

Secure references cause DCOM to make extra callbacks to insure that objects are not released maliciously.

See Also

RPC_C_IMP_LEVEL_*xxx*, **RPC_C_AUTHN_LEVEL_*xxx***, **Security in COM**

CoIsHandlerConnected Quick Info

Determines whether a remote object is connected to the corresponding in-process object.

```
BOOL CoIsHandlerConnected(
  LPUNKNOWN pUnk      //Pointer to the remote object
);
```

Parameter

pUnk

[in] Pointer to the controlling **IUnknown** interface on the remote object.

Return Values

TRUE

The object is not remote or that it is remote and is still connected to its remote handler.

FALSE

The object is remote and is invalid (no longer connected to its remote handler).

Remarks

The **CoIsHandlerConnected** function determines the status of a remote object. You can use it to determine when to release a remote object. You specify the remote object by giving the function a pointer to its controlling **IUnknown** interface (the *pUnk* parameter). A TRUE returned from the function indicates either that the specified object is not remote, or that it is remote and is still connected to its remote handler. A FALSE returned from the function indicates that the object is remote but is no longer connected to its remote handler; in this case, the caller should respond by releasing the object.

CoIsOle1Class Quick Info

Determines if a given CLSID represents an OLE 1 object.

```
BOOL CoIsOle1Class(

  REFCLSID rclsid      //CLSID to check
);
```

Parameter

rclsid

[in] CLSID to check.

Return Values

S_TRUE

CLSID refers to an OLE 1 object.

S_FALSE

CLSID does not refer to an OLE 1 object.

Remarks

CoIsOle1Class determines whether an object class is from OLE 1. You can use it to prevent linking to embedded OLE 1 objects within a container, which OLE 1 objects do not support. Once a container has determined that copied data represents an embedded object, the container code can call **CoIsOle1Class** to determine whether the embedded object is an OLE 1 object. If **CoIsOle1Class** returns S_TRUE, the container does not offer CF_LINKSOURCE as one of its clipboard formats. This is one of several OLE compatibility functions. Other compatibility functions, listed below, can be used to convert the storage formats of objects between OLE 1 and OLE.

See Also

OleConvertIStorageToOLESTREAM, **OleConvertIStorageToOLESTREAMEx**, **OleConvertOLESTREAMToIStorage**, **OleConvertOLESTREAMToIStorageEx**

CoLoadLibrary Quick Info

Loads a specific DLL into the caller's process. The **CoGetClassObject** function calls **CoLoadLibrary** internally; applications should not call it directly.

```
HINSTANCE CoLoadLibrary(

  LPOLESTR lpszLibName,     //Pointer to the name of the library to be loaded
  BOOL bAutoFree            //Whether library is automatically freed
);
```

Parameters

lpszLibName

[in] Pointer to the name of the library to be loaded. The use of this name is the same as in the Win32 function **LoadLibrary**.

bAutoFree

[in] If TRUE, indicates that this library is freed when it is no longer needed, through a call to either the **CoFreeUnusedLibraries** or **CoUninitialize** functions. If FALSE, the library should be explicitly freed with the **CoFreeLibrary** function.

Return Values

Module Handle

Handle of the loaded library.

NULL

Library could not be loaded.

Remarks

The **CoLoadLibrary** function is called internally by the **CoGetClassObject** function when the class context (CLSCTX) indicates a DLL. **CoLoadLibrary** loads a DLL specified by the *lpszLibName* parameter into the process that called **CoGetClassObject**. Containers should not call **CoLoadLibrary** directly.

Internally, a reference count is kept on the loaded DLL, by using **CoLoadLibrary** to increment the count and the **CoFreeLibrary** function to decrement it.

See Also

CoFreeAllLibraries, **CoFreeLibrary**, **CoFreeUnusedLibraries**, **CoGetClassObject**

CoLockObjectExternal Quick Info

Called either to lock an object to ensure that it stays in memory, or to release such a lock. Call **CoLockObjectExternal** to place a strong lock on an object to ensure that it stays in memory.

```
STDAPI CoLockObjectExternal(
  IUnknown * pUnk,              //Pointer to object to be locked or unlocked
  BOOL fLock,                   //TRUE = lock, FALSE = unlock
  BOOL fLastUnlockReleases      //TRUE = release all pointers to object
);
```

Parameters

pUnk

[in] Pointer to the **IUnknown** interface on the object to be locked or unlocked.

fLock

[in] Whether the object is to be locked or released. Specifying TRUE holds a reference to the object (keeping it in memory), locking it independently of external or internal **AddRef/Release** operations, registrations, or revocations. If *fLock* is TRUE, *fLastLockReleases* is ignored. FALSE releases a lock previously set with a call to this function.

fLastUnlockReleases

[in] Whether a given lock is the last reference that is supposed to keep an object alive. If it is, TRUE releases all pointers to the object (there may be other references that are not supposed to keep it alive).

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The object was locked successfully.

Remarks

The **CoLockObjectExternal** function prevents the reference count of an object from going to zero, thereby "locking" it into existence until the lock is released. The same function (with different parameters) releases the lock. The lock is implemented by having the system call **IUnknown::AddRef** on the object. The system then waits to call **IUnknown::Release** on the object until a later call to **CoLockObjectExternal** with *fLock* set to FALSE. This function can be used to maintain a reference count on the object on behalf of the end user, because it acts outside of the object, as does the user.

The **CoLockObjectExternal** function *must* be called in the process in which the object actually resides (the EXE process, not the process in which handlers may be loaded).

Calling **CoLockObjectExternal** sets a strong lock on an object. A strong lock keeps an object in memory, while a weak lock does not. Strong locks are required, for example, during a silent update to an OLE embedding. The embedded object's container must remain in memory until the update process is complete. There must also be a strong lock on an application object to ensure that the application stays

alive until it has finished providing services to its clients. All external references place a strong reference lock on an object.

The **CoLockObjectExternal** function is typically called in the following situations:

- Object servers should call **CoLockObjectExternal** with both *fLock* and *fLastLockReleases* set to TRUE when they become visible. This call creates a strong lock on behalf of the user. When the application is closing, free the lock with a call to **CoLockObjectExternal**, setting *fLock* to FALSE and *fLastLockReleases* to TRUE.
- A call to **CoLockObjectExternal** on the server can also be used in the implementation of **IOleContainer::LockContainer**.

There are several things to be aware of when you use **CoLockObjectExternal** in the implementation of **IOleContainer::LockContainer**. An embedded object would call **IOleContainer::LockContainer** on its container to keep it running (to lock it) in the absence of other reasons to keep it running. When the embedded object becomes visible, the container must weaken its connection to the embedded object with a call to the **OleSetContainedObject** function, so other connections can affect the object.

Unless an application manages all aspects of its application and document shutdown completely with calls to **CoLockObjectExternal**, the container must keep a private lock count in **IOleContainer::LockContainer** so that it exits when the lock count reaches zero and the container is invisible. Maintaining all aspects of shutdown, and thereby avoiding keeping a private lock count, means that **CoLockObjectExternal** should be called whenever one of the following conditions occur:

- A document is created and destroyed or made visible or invisible.
- An application is started and shut down by the user.
- A pseudo-object is created and destroyed.

For debugging purposes, it may be useful to keep a count of the number of external locks (and unlocks) set on the application.

Note The end user has explicit control over the lifetime of an application, even if there are external locks on it. That is, if a user decides to close the application (File, Exit), it *must* shut down. In the presence of external locks (such as the lock set by **CoLockObjectExternal**), the application can call the **CoDisconnectObject** function to force these connections to close prior to shutdown.

See Also

IOleContainer::LockContainer, **OleSetContainedObject**

CoMarshalHresult Quick Info

Marshals an HRESULT to the specified stream, from which it can be unmarshaled using the **CoUnmarshalHresult** function.

```
STDAPI CoMarshalHresult(
  IStream * pStm,      //Pointer to the marshaling stream
  HRESULT hresult      //HRESULT to be marshaled
);
```

Parameters

pStm

[in] Pointer to the marshaling stream.

hresult

[in] HRESULT in the originating process.

Return Values

This function supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The HRESULT was marshaled successfully.

STG_E_INVALIDPOINTER

Bad pointer passed in for *pStm*.

STG_E_MEDIUMFULL

The medium is full.

Remarks

An HRESULT is process-specific, so an HRESULT that is valid in one process might not be valid in another. If you are writing your own implementation of **IMarshal** and need to marshal an HRESULT from one process to another, either as a parameter or a return code, you must call this function. In other circumstances, you will have no need to call this function.

This function perfoms the following tasks:

1. Writes an HRESULT to a stream.
2. Returns an **IStream** pointer to that stream.

See Also

CoUnmarshalHresult, **IStream**

CoMarshalInterface Quick Info

Writes into a stream the data required to initialize a proxy object in some client process. The COM library in the client process calls the **CoUnmarshalInterface** function to extract the data and initialize the proxy. **CoMarshalInterface** can marshal only interfaces derived from **IUnknown**.

```
STDAPI CoMarshalInterface(
    IStream *pStm,          //Pointer to the stream used for marshaling
    REFIID riid,            //Reference to the identifier of the interface
    IUnknown *pUnk,         //Pointer to the interface to be marshaled
    DWORD dwDestContext,    //Destination context
    void *pvDestContext,    //Reserved for future use
    DWORD mshlflags         //Reason for marshaling
);
```

Parameters

pStm

[in] Pointer to the stream to be used during marshaling.

riid

[in] Reference to the identifier of the interface to be marshaled. This interface must be derived from the **IUnknown** interface.

pUnk

[in] Pointer to the interface to be marshaled; can be NULL if the caller does not have a pointer to the desired interface. This interface must be derived from the **IUnknown** interface.

dwDestContext

[in] Destination context where the specified interface is to be unmarshaled. Values for *dwDestContext* come from the enumeration **MSHCTX**. Currently, unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext

[in] Reserved for future use; must be NULL.

mshlflags

[in] Flag specifying whether the data to be marshaled is to be transmitted back to the client process—the normal case—or written to a global table, where it can be retrieved by multiple clients. Values come from the **MSHLFLAGS** enumeration.

Return Values

This function supports the standard return values E_FAIL, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The interface pointer was marshaled successfully.

CO_E_NOTINITIALIZED

The **CoInitialize** or **OleInitialize** function was not called on the current thread before this function was called.

IStream errors

This function can also return any of the stream-access error values returned by the **IStream** interface.

Remarks

The **CoMarshalInterface** function marshals the interface referred to by *riid* on the object whose **IUnknown** implementation is pointed to by *pUnk*. To do so, the **CoMarshalInterface** function performs the following tasks:

1. Queries the object for a pointer to the **IMarshal** interface. If the object does not implement **IMarshal**, meaning that it relies on COM to provide marshaling support, **CoMarshalInterface** gets a pointer to COM's default implementation of **IMarshal**.
2. Gets the CLSID of the object's proxy by calling **IMarshal::GetUnmarshalClass**, using whichever **IMarshal** interface pointer has been returned.
3. Writes the CLSID of the proxy to the stream to be used for marshaling.
4. Marshals the interface pointer by calling **IMarshal::MarshalInterface**.

If you are implementing existing COM interfaces or defining your own interfaces using the Microsoft Interface Definition Language (MIDL), the MIDL-generated proxies and stubs call **CoMarshalInterface** for you. If you are writing your own proxies and stubs, your proxy code and stub code should each call **CoMarshalInterface** to correctly marshal interface pointers. Calling **IMarshal** directly from your proxy and stub code is not recommended.

If you are writing your own implementation of **IMarshal**, and your proxy needs access to a private object, you can include an interface pointer to that object as part of the data you write to the stream. In such situations, if you want to use COM's default marshaling implementation when passing the interface pointer, you can call **CoMarshalInterface** on the object to do so.

See Also

CoUnmarshalInterface, **IMarshal::MarshalInterface**

CoMarshalInterThreadInterfaceInStream

Marshals an interface pointer from one thread to another thread in the same process.

```
HRESULT CoMarshalInterThreadInterfaceInStream(
  REFIID riid,          //Reference to the identifier of the interface
  LPUNKNOWN pUnk,       //Pointer to the interface to be marshaled
  LPSTREAM * ppStm      //Indirect pointer
);
```

Parameters

riid

[in] Reference to the identifier of the interface to be marshaled.

pUnk

[in] Pointer to the interface to be marshaled, which must be derived from **IUnknown**; can be NULL.

ppStm

[out] Indirect pointer to the stream that contains the marshaled interface.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The interface was marshaled successfully.

Remarks

The **CoMarshalInterThreadInterfaceInStream** function enables an object to easily and reliably marshal an interface pointer to another thread in the same process. The stream returned in *ppStm* is guaranteed to behave correctly when a client running in the receiving thread attempts to unmarshal the pointer. The client can then call the **CoGetInterfaceAndReleaseStream** to unmarshal the interface pointer and release the stream object.

The **CoMarshalInterThreadInterfaceInStream** function performs the following tasks:

1. Creates a stream object.
2. Passes the stream object's **IStream** pointer to **CoMarshalInterface**.
3. Returns the **IStream** pointer to the caller.

See Also

CoGetInterfaceAndReleaseStream

CoQueryAuthenticationServices Quick Info

Retrieves a list of the authentication services registered when the process called **CoInitializeSecurity**.

```
HRESULT CoQueryAuthenticationServices(

  DWORD * pcAuthSvc,                          //Pointer to the number of entries returned in the array
  SOLE_AUTHENTICATION_SERVICE**  prgAuthSvc   //Pointer to an array of structures
);
```

Parameters

pcAuthSvc

[out] Pointer to return the number of entries returned in the *rgAuthSvc* array. May not be NULL.

prgAuthSvc

[out] Pointer to an array of **SOLE_AUTHENTICATION_SERVICE** structures. The list is allocated through a call to **CoTaskMemAlloc**. The caller must free the list when finished with it by calling **CoTaskMemFree**.

Return Values

This function supports the standard return values E_INVALIDARG and

E_OUTOFMEMORY, as well as the following:

S_OK

Indicates success.

Remarks

CoQueryAuthenticationServices retrieves a list of the authentication services currently registered. If the process calls **CoInitializeSecurity**, these are the services registered through that call; if not, those registered by default by OLE.

This function is primarily useful for custom marshalers, to determine which principal names an application can use.

Different authentication services support different levels of security. For example, NTLMSSP does not support delegation or mutual authentication while *Kerberos* does. The application is responsible only for registering authentication services that provide the features the application needs. This is the way to query which services have been registered with **CoInitializeSecurity**.

See Also

CoInitializeSecurity, **SOLE_AUTHENTICATION_SERVICE** structure, **Security in COM**

CoQueryClientBlanket Quick Info

Called by the server to find out about the client that invoked the method executing on the current thread.

```
HRESULT CoQueryClientBlanket(
    DWORD*    pAuthnSvc,          //Pointer to the current authentication service
    DWORD*    pAuthzSvc,          //Pointer to the current authorization service
    OLECHAR **    pServerPrincName,   //Pointer to the current principal name
    DWORD * pAuthnLevel,          //Pointer to the current authentication level
    DWORD * pImpLevel,            //Must be NULL
    void **   ppPrivs,            //Pointer to unicode string identifying client
    DWORD **  pCapabilities       //Pointer to flags indicating further capabilities of the proxy
);
```

Parameters

pAuthnSvc

[out] Pointer to a DWORD value defining the current *authentication* service. This will be a single value taken from the list of **RPC_C_AUTHN_*xxx*** constants. May be NULL, in which case the current authentication service is not returned.

pAuthzSvc

[out] Pointer to a DWORD value defining the current *authorization* service. This will be a single value taken from the list of **RPC_C_AUTHZ_*xxx*** constants. May be NULL, in which case the current authorization service is not returned.

pServerPrincName

[out] Pointer to the current principal name. The string will be allocated by the callee using **CoTaskMemAlloc** and must be freed by the caller using **CoTaskMemFree** when they are done with it. May be NULL, in which case the principal name is not returned.

pAuthnLevel

[out] Pointer to a DWORD value defining the current authentication level. This will be a single value taken from the list of **RPC_C_AUTHN_LEVEL_*xxx*** constants. May be NULL, in which case the current authentication level is not returned.

pImpLevel

[out] Must be NULL; does not supplythe current impersonation level is not returned.

pPrivs

[out] Pointer to a unicode string identifying the client. This string is not a copy, the user must not change it or free it. The string is not valid past the end of the call. For NTLMSSP the string is of the form *domain\user*.

pCapabilities

[out] Pointer to flags indicating further capabilities of the call. Currently, no flags are defined for this parameter and the value retrieved is EOAC_NONE. May be NULL, in which case no value is retrieved. EOAC_MUTUAL_AUTH is defined but not used by NTLMSSP. Third party security providers may return that flag.

Return Values

S_OK

Success.

E_INVALIDARG

One or more arguments are invalid.

E_OUTOFMEMORY

Insufficient memory to create the *pServerPrincName* out-parameter.

Remarks

CoQueryClientBlanket is called by the server to get security information about the client that invoked the method executing on the current thread. This function encapsulates the following sequence of common calls (error handling excluded):

```
CoGetCallContext(IID_IServerSecurity, (void**)&pss);
pss->QueryBlanket(pAuthnSvc, pAuthzSvc, pServerPrincName,
            pAuthnLevel, pImpLevel, pPrivs, pCapabilities);
pss->Release();
```

See Also

IServerSecurity::QueryBlanket, **Security in COM**

CoQueryProxyBlanket Quick Info

Retrieves the authentication information the client uses to make calls on the specified proxy.

```
HRESULT CoQueryProxyBlanket(
  void*   pProxy,                          //Location for the proxy to query
  DWORD*   pAuthnSvc,                      //Location for the the current authorization service
  DWORD*   pAuthzSvc,                      //Location for the the current authorization service
  OLECHAR **   pServerPrincName,           //Location for the current principal name
  DWORD * pAuthnLevel,                     //Location for the current authentication level
  DWORD * pImpLevel,                       //Location for the current impersonation level
  RPC_AUTH_IDENTITY_HANDLE **   ppAuthInfo,//Location for the value passed to IClientSecurity::SetBlanket
  DWORD **   pCapabilities                 //Location for flags indicating further capabilities of the proxy
);
```

Parameters

pProxy

[in] Pointer to an interface on the proxy to query.

pAuthnSvc

[out] Pointer to a DWORD value defining the current *authentication* service. This will be a single value taken from the list of **RPC_C_AUTHN_*xxx*** constants. May be NULL, in which case the current authentication service is not retrieved.

pAuthzSvc

[out] Pointer to a DWORD value defining the current *authorization* service. This will be a single value taken from the list of **RPC_C_AUTHZ_*xxx*** constants. May be NULL, in which case the current authorization service is not retrieved.

pServerPrincName

[out] Pointer to the current principal name. The string will be allocated by the one called using **CoTaskMemAlloc** and must be freed by the caller using **CoTaskMemFree** when they are done with it. May be NULL, in which case the principal name is not retrieved.

pAuthnLevel

[out] Pointer to a DWORD value defining the current authentication level. This will be a single value taken from the list of **RPC_C_AUTHN_LEVEL_*xxx*** constants. May be NULL, in which case the current authentication level is not retrieved.

pImpLevel

[out] Pointer to a DWORD value defining the current impersonation level. This will be a single value taken from the list of **RPC_C_IMP_LEVEL_*xxx*** constants. May be NULL, in which case the current authentication level is not retrieved.

ppAuthInfo

[out] Pointer to the pointer value passed to **IClientSecurity::SetBlanket** indicating the identity of the client. Because this points to the value itself and is not a copy, it should not be manipulated. May be NULL, in which case the information is not retrieved.

pCapabilities

[out] Pointer to a DWORD of flags indicating further capabilities of the proxy. Currently, no flags are defined for this parameter and it will only return EOAC_NONE. May be NULL, in which case the flags indicating further capabilities are not retrieved.

Return Values

S_OK

Success.

E_INVALIDARG

One or more arguments are invalid.

E_OUTOFMEMORY

Insufficient memory to create the *pasAuthnSvc* out-parameter.

Remarks

CoQueryProxyBlanket is called by the client to retrieve the authentication information COM will use on calls made from the specified proxy. This function encapsulates the following sequence of common calls (error handling excluded):

```
pProxy->QueryInterface(IID_IClientSecurity, (void**)&pcs);
pcs->QueryBlanket(pProxy, pAuthnSvc, pAuthzSvc, pServerPrincName,
                  pAuthnLevel, pImpLevel, ppAuthInfo, pCapabilities);
pcs->Release();
```

This sequence calls **QueryInterface** on the proxy for **IClientSecurity**, and with the resulting pointer, calls **IClientSecurity::QueryBlanket**, and then releases the pointer.

In *pProxy*, you can pass any proxy, such as a proxy you get through a call to **CoCreateInstance**, **CoUnmarshalInterface**, or just passing an interface pointer as a parameter. It can be any interface. You cannot pass a pointer to something that is not a proxy. Thus you can't pass a pointer to an interface that has the local keyword in its interface definition since no proxy is created for such an interface. **IUnknown** is the exception.

See Also

IClientSecurity::QueryBlanket, **Security in COM**

CoRegisterClassObject Quick Info

Registers an EXE class object with OLE so other applications can connect to it. EXE object applications should call **CoRegisterClassObject** on startup. It can also be used to register internal objects for use by the same EXE or other code (such as DLLs) that the EXE uses.

```
STDAPI CoRegisterClassObject(
  REFCLSID rclsid,          //Class identifier (CLSID) to be registered
  IUnknown * pUnk,          //Pointer to the class object
  DWORD dwClsContext,       //Context for running executable code
  DWORD flags,              //How to connect to the class object
  LPDWORD * lpdwRegister    //Pointer to the value returned
);
```

Parameters

rclsid

[in] CLSID to be registered.

pUnk

[in] Pointer to the**IUnknown** interface on the class object whose availability is being published.

dwClsContext

[in] Context in which the executable code is to be run. For information on these context values, see the **CLSCTX** enumeration.

flags

[in] How connections are made to the class object. For information on these flags, see the **REGCLS** enumeration.

lpdwRegister

[out] Pointer to a value that identifies the class object registered; later used by the **CoRevokeClassObject** function to revoke the registration.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The class object was registered successfully.

CO_E_OBJISREG

Already registered in the class object table.

Remarks

Only EXE object applications call **CoRegisterClassObject**. Object handlers or DLL object applications do not call this function – instead, they must implement and export the **DllGetClassObject** function.

At startup, a multiple-use EXE object application must create a class object (with the **IClassFactory** interface on it), and call **CoRegisterClassObject** to register the class object. Object applications that support several different classes (such as multiple types of embeddable objects) must allocate and register a different class object for each.

Multiple registrations of the same class object are independent and do not produce an error. Each subsequent registration yields a unique key in *lpdwRegister*.

Multiple document interface (MDI) applications must register their class objects. Single document interface (SDI) applications must register their class objects only if they can be started by means of the **/Embedding** switch.

The server for a class object should call **CoRevokeClassObject** to revoke the class object (remove its registration) when all of the following are true:

- There are no existing instances of the object definition
- There are no locks on the class object
- The application providing services to the class object is not under user control (not visible to the user on the display).

After the class object is revoked, when its reference count reaches zero, the class object can be released, allowing the application to exit.

For information on the *flags* parameter, refer to the **REGCLS** enumeration.

See Also

CoGetClassObject, **CoRevokeClassObject**, **DllGetClassObject**, **REGCLS**, **CLSCTX**

CoRegisterMallocSpy Quick Info

Registers an implementation of the **IMallocSpy** interface in OLE, thereafter requiring OLE to call its wrapper methods around every call to the corresponding **IMalloc** method. **IMallocSpy** is defined in OLE to allow developers to debug memory allocations.

```
HRESULT CoRegisterMallocSpy(

  LPMALLOCSPY pMallocSpy      //Pointer to the interface
);
```

Parameter

pMallocSpy

[in] Pointer to an instance of the **IMallocSpy** implementation.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The **IMallocSpy** object is successfully registered.

CO_E_OBJISREG

There is already a registered spy.

Remarks

The **CoRegisterMallocSpy** function registers the **IMallocSpy** object, which is used to debug calls to **IMalloc** methods. The function calls **QueryInterface** on the pointer *pMallocSpy* for the interface **IID_IMallocSpy**. This is to ensure that *pMallocSpy* really points to an implementation of **IMallocSpy**. By the rules of OLE, it is expected that a successful call to **QueryInterface** has added a reference (through the **AddRef** method) to the **IMallocSpy** object. That is, **CoRegisterMallocSpy** does not directly call **AddRef** on *pMallocSpy*, but fully expects that the **QueryInterface** call will.

When the **IMallocSpy** object is registered, whenever there is a call to one of the **IMalloc** methods, OLE first calls the corresponding **IMallocSpy** pre-method. Then, after executing the **IMalloc** method, OLE calls the corresponding **IMallocSpy** post-method. For example, whenever there is a call to **IMalloc::Alloc**, from whatever source, OLE calls **IMallocSpy::PreAlloc**, calls **IMalloc::Alloc**, and after that allocation is completed, calls **IMallocSpy::PostAlloc**.

See Also

IMallocSpy, **CoRevokeMallocSpy**, **CoGetMalloc**

CoRegisterMessageFilter Quick Info

Registers with OLE the instance of an EXE application's **IMessageFilter** interface, which is to be used for handling concurrency issues. DLL object applications cannot register a message filter.

```
HRESULT CoRegisterMessageFilter(
  LPMESSAGEFILTER lpMessageFilter,      //Pointer to interface
  LPMESSAGEFILTER * lplpMessageFilter   //Indirect pointer to prior instance if non-NULL
);
```

Parameters

lpMessageFilter

[in] Pointer to the**IMessageFilter** interface on the message filter supplied by the application. Can be NULL, indicating that the current **IMessageFilter** registration should be revoked.

lplpMessageFilter

[out] If a message filter has been previously registered, indirect pointer to an **IMessageFilter** interface to that instance. If the value of this parameter on return is NULL, this indicates that no previous **IMessageFilter** instance was registered. This parameter rarely returns NULL, however, returning instead a pointer to the default message filter.

Return Values

S_OK

The **IMessageFilter** instance registered or revoked successfully.

S_FALSE

Error registering or revoking **IMessageFilter** instance.

CoRegisterPSClsid

Enables a downloaded DLL to register its custom interfaces within its running process so that the marshaling code will be able to marshal those interfaces.

```
WINOLEAPI CoRegisterPSCLsid(
    REFIID riid,                // Custom interface to be registered
    REFCLSID rclsid             // DLL containing the proxy/stub code for riid
);
```

Parameters

riid

[in] Points to the IID of the interface to be registered.

rclsid

[in] Points to the CLSID of the DLL that contains the proxy/stub code for the custom interface specified by *riid*.

Return Values

S_OK

The custom interface was successfully registered.

E_INVALIDARG

One of the parameters is invalid.

E_OUTOFMEMORY

There is insufficient memory to complete this operation.

Remarks

Normally the code responsible for marshaling an interface pointer into the current running process reads the HKEY_CLASSES_ROOT\Interfaces section of the registry to obtain the CLSID of the DLL containing the ProxyStub code to be loaded. To obtain the ProxyStub CLSIDs for an existing interface, the code calls the **CoGetPSClsid** function.

In some cases, however, it may be desirable or necessary for an in-process handler or in-process server to make its custom interfaces available without writing to the registry. A DLL downloaded across a network may not even have permission to access the local registry, and because the code originated on another machine, the user, for security purposes, may want to run it in a restricted environment. Or a DLL may have custom interfaces that it uses to talk to a remote server and may also include the ProxyStub code for those interfaces. In such cases, a DLL needs an alternative way to register its interfaces. **CoRegisterPSClsid**, used in conjunction with **CoRegisterClassObject**, provides that alternative.

A DLL would normally call **CoRegisterPSClsid** as shown in the following code fragment:

```
HRESULT RegisterMyCustomInterface(DWORD *pdwRegistrationKey)
{
    HRESULT hr = CoRegisterClassObject(CLSID_MyProxyStubClsid,
        pIPSFactoryBuffer,
        CLSCTX_INPROC_SERVER,
        REGCLS_MULTIPLEUSE
        pdwRegistrationKey);
    if(SUCCEEDED)(hr))
    {
        hr = CoRegisterPSClsid(IID_MyCustomInterface,
CLSID_MyProxyStubClsid);
    }

    return hr;
}
```

See Also

CoGetPSClsid, CoRegisterClassObject

CoReleaseMarshalData Quick Info

Destroys a previously marshaled data packet.

```
STDAPI CoReleaseMarshalData(

  IStream * pStm      //Pointer to stream containing data packet
);
```

Parameter

pStm

[in] Pointer to the stream that contains the data packet to be destroyed.

Return Values

This function supports the standard return values E_FAIL, E_INVALIDARG,

E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The data packet was successfully destroyed.

STG_E_INVALIDPOINTER

An **IStream** error dealing with the *pStm* parameter.

CO_E_NOTINITIALIZED

The **CoInitialize** or **OleInitialize** function was not called on the current thread before this function was called.

Remarks

The **CoReleaseMarshalData** function performs the following tasks:

1. The function reads a CLSID from the stream.
2. If COM's default marshaling implementation is being used, the function gets an **IMarshal** pointer to an instance of the standard unmarshaler. If custom marshaling is being used, the function creates a proxy by calling the **CoCreateInstance** function, passing the CLSID it read from the stream, and requesings an **IMarshal** interface pointer to the newly created proxy.
3. Using whichever **IMarshal** interface pointer it has acquired, the function calls **IMarshal::ReleaseMarshalData**.

You typically do not call this function. The only situation in which you might need to call this function is if you use custom marshaling (write and use your own implementation of **IMarshal**). Examples of when **CoReleaseMarshalData** should be called include the following situations:

- An attempt was made to unmarshal the data packet, but it failed.
- A marshaled data packet was removed from a global table.

As an analogy, the data packet can be thought of as a reference to the original object, just as if it were another interface pointer being held on the object. Like a real interface pointer, that data packet must be released at some point. The use of **IMarshal::ReleaseMarshalData** to release data packets is analogous to the use of **IUnknown::Release** to release interface pointers.

Note that you do not need to call **CoReleaseMarshalData** after a successful call of the **CoUnmarshalInterface** function; that function releases the marshal data as part of the processing that it does.

See Also

IMarshal::ReleaseMarshalData

CoReleaseServerProcess

Decrements the global per-process reference count

ULONG CoReleaseServerProcess(void**);**

Return Values

S_OK

The CLSID was retrieved successfully.

Remarks

Servers can call **CoReleaseServerProcess** to decrement a global per-process reference count incremented through a call to **CoAddRefServerProcess**

When that count reaches zero, OLE automatically calls **CoSuspendClassObjects**, which prevents new activation requests from coming in. This permits the server to deregister its class objects from its various threads without worry that another activation request may come in. New activation requests result in launching a new instance of the local server process.

The simplest way for a local server application to make use of these API functions is to call **CoAddRefServerProcess** in the constructor for each of its instance objects, and in each of its **IClassFactory::LockServer** methods when the *fLock* parameter is TRUE. The server application should also call **CoReleaseServerProcess** in the destruction of each of its instance objects, and in each of its **IClassFactory::LockServer** methods when the *fLock* parameter is FALSE. Finally, the server application must check the return code from **CoReleaseServerProcess**; if it returns 0, the server application should initiate its cleanup. This typically means that a server with multiple threads should signal its various threads to exit their message loops and call **CoRevokeClassObject** and **CoUninitialize**.

If these APIs are used at all, they must be called in both the object instances and the **LockServer** method, otherwise the server application may be shutdown prematurely. In-process Servers typically should not call **CoAddRefServerProcess** or **CoReleaseServerProcess**.

See Also

CoSuspendClassObjects, **CoReleaseServerProcess**, **IClassFactory::LockServer**, **Out-of-process Server Implementation Helpers**

CoResumeClassObjects

Called by a server that can register multiple class objects to inform the OLE SCM about all registered classes, and permits activation requests for those class objects.

WINOLEAPI CoResumeClassObjects(void**);**

Return Values

S_OK

The CLSID was retrieved successfully.

Remarks

Servers that can register multiple class objects call **CoResumeClassObjects** once, after having first called **CoRegisterClassObject**, specifying REGCLS_LOCAL_SERVER | REGCLS_SUSPENDED for each CLSID the server supports. This function causes OLE to inform the SCM about all the registered classes, and begins letting activation requests into the server process.

This reduces the overall registration time, and thus the server application startup time, by making a single call to the SCM, no matter how many CLSIDs are registered for the server. Another advantage is that if the server has multiple apartments with different CLSIDs registered in different apartments, or is a free-threaded server, no activation requests will come in until the server calls **CoResumeClassObjects**. This gives the server a chance to register all of its CLSIDs and get properly set up before having to deal with activation requests, and possibly shutdown requests.

See Also

CoRegisterClassObject, **CoSuspendClassObjects**, **Out-of-process Server Implementation Helpers**

CoRevertToSelf Quick Info

Restores the authentication information on a thread of execution to its previous identity.

HRESULT CoRevertToSelf()

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Indicates success.

Remarks

CoRevertToSelf restores the authentication information on a thread of execution to its previous identity after a previous server call to **CoImpersonateClient**. This is a helper function that encapsulates the following common sequence of calls (error handling excluded):

```
CoGetCallContext(IID_IServerSecurity, (void**)&pss);
pss->RevertToSelf();
pss->Release();
```

See Also

IServerSecurity::RevertToSelf, **CoGetCallContext**, **Security in COM**

CoRevokeClassObject Quick Info

Informs OLE that a class object, previously registered with the **CoRegisterClassObject** function, is no longer available for use.

```
HRESULT CoRevokeClassObject(
  DWORD dwRegister      //Token on class object
);
```

Parameter

dwRegister

[in] Token previously returned from the **CoRegisterClassObject** function.

Return Values

This function supports the standard return values E_INVALIDARG,

E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The class object was successfully revoked.

Remarks

A successful call to **CoRevokeClassObject** means that the class object has been removed from the global class object table (although it does not release the class object). If other clients still have pointers to the class object and have caused the reference count to be incremented by calls to **IUnknown::AddRef**, the reference count will not be zero. When this occurs, applications may benefit if subsequent calls (with the obvious exceptions of **IUnknown::AddRef** and **IUnknown::Release**) to the class object fail.

An object application *must* call **CoRevokeClassObject** to revoke registered class objects before exiting the program. Class object implementers should call **CoRevokeClassObject** as part of the release sequence. You must specifically revoke the class object even when you have specified the *flags* value REGCLS_SINGLEUSE in a call to **CoRegisterClassObject**, indicating that only one application can connect to the class object.

See Also

CoGetClassObject, **CoRegisterClassObject**

CoRevokeMallocSpy Quick Info

Revokes a registered **IMallocSpy** object.

HRESULT CoRevokeMallocSpy();

Return Values

S_OK

The **IMallocSpy** object is successfully revoked.

CO_E_OBJNOTREG

No spy is currently registered.

E_ACCESSDENIED

Spy is registered but there are outstanding allocations (not yet freed) made while this spy was active.

Remarks

The **IMallocSpy** object is released when it is revoked. This release corresponds to the call to **IUnknown::AddRef** in the implementation of the **QueryInterface** function by the **CoRegisterMallocSpy** function. The implementation of the **IMallocSpy** interface should then do any appropriate cleanup.

If the return code is E_ACCESSDENIED, there are still outstanding allocations that were made while the spy was active. In this case, the registered spy cannot be revoked at this time because it may have attached arbitrary headers and/or trailers to these allocations that only the spy knows about. Only the spy's **PreFree** (or **PreRealloc**) method knows how to account for these headers and trailers. Before returning E_ACCESSDENIED, **CoRevokeMallocSpy** notes internally that a revoke is pending. When the outstanding allocations have been freed, the revoke proceeds automatically, releasing the **IMallocSpy** object. Thus, it is necessary to call **CoRevokeMallocSpy** only once for each call to **CoRegisterMallocSpy**, even if E_ACCESSDENIED is returned.

See Also

IMallocSpy, **CoRegisterMallocSpy**, **CoGetMalloc**

CoSetProxyBlanket Quick Info

Sets the authentication information that will be used to make calls on the specified proxy.

```
HRESULT CoSetProxyBlanket(
  void *   pProxy,                          //Indicates the proxy to set
  DWORD    dwAuthnSvc,                      //Authentication service to use
  DWORD    dwAuthzSvc,                      //Authorization service to use
  WCHAR *  pServerPrincName,                //The server principal name to use with the authentication service
  DWORD    dwAuthnLevel,                    //The authentication level to use
  DWORD    dwImpLevel,                      //The impersonation level to use
  RPC_AUTH_IDENTITY_HANDLE *   pAuthInfo,   //The identity of the client
  DWORD    dwCapabilities                   //Undefined – capability flags
);
```

Parameter

pProxy

[in] Pointer to an interface on the proxy for which this authentication information is to be set.

dwAuthnSvc

[in] A single DWORD value from the list of **RPC_C_AUTHN_*xxx*** constants indicating the *authentication* service to use. It may be RPC_C_AUTHN_NONE if no authentication is required.

dwAuthzSvc

[in] A single DWORD value from the list of **RPC_C_AUTHZ_*xxx*** constants indicating the *authorization* service to use. If you are using the system default authentication service, use RPC_C_AUTHZ_NONE.

pServerPrincName

[in] Points to a WCHAR string that indicates the server principal name to use with the authentication service. If you are using RPC_C_AUTHN_WINNT, the principal name must be NULL.

dwAuthnLevel

[in] A single DWORD value from the list of **RPC_C_AUTHN_LEVEL_*xxx*** constants indicating the authentication level to use.

dwImpLevel

[in] A single DWORD value from the list of **RPC_C_IMP_LEVEL_*xxx*** constants indicating the impersonation level to use. Currently, only RPC_C_IMP_LEVEL_IMPERSONATE and RPC_C_IMP_LEVEL_IDENTIFY are supported.

pAuthInfo

[in] Establishes the identity of the client. It is authentication service specific. Some authentication services allow the application to pass in a different user name and password. COM keeps a pointer to the memory passed in until COM is uninitialized or a new value is set. If NULL is specified COM uses the current identity (the process token). For NTLMSSP the structure is SEC_WINNT_AUTH_IDENTITY_W.

dwCapabilities

[in] Flags to establish indicating the further capabilities of this proxy. Currently, no capability flags are

defined.

The caller should specify EOAC_NONE. EOAC_MUTUAL_AUTH is defined and may be used by other security providers, but is not supported by NTLMSSP. Thus, NTLMSSP will accept this flag without generating an error but without providing mutual authentication.

Return Values

S_OK

Success, append the headers.

E_INVALIDARG

One or more arguments is invalid.

Remarks

Sets the authentication information that will be used to make calls on the specified proxy. This function encapsulates the following sequence of common calls (error handling excluded):

```
pProxy->QueryInterface(IID_IClientSecurity, (void**)&pcs);
pcs->SetBlanket(pProxy, dwAuthnSvc, dwAuthzSvc, pServerPrincName,
    dwAuthnLevel, dwImpLevel, pAuthInfo, dwCapabilities);
pcs->Release();
```

See Also

IClientSecurity::SetBlanket, **CoQueryClientBlanket**, **Security in COM**

CoSuspendClassObjects

Prevents any new activation requests from the SCM on all class objects registered within the process.

WINOLEAPI CoSuspendClassObjects(void**);**

Return Values

S_OK

The CLSID was retrieved successfully.

Remarks

CoSuspendClassObjects prevents any new activation requests from the SCM on all class objects registered within the process. Even though a process may call this API, the process still must call **CoRevokeClassObject** for each CLSID it has registered, in the apartment it registered in. Applications typically do not need to call this API, which is generally only called internally by OLE when used in conjunction with **CoReleaseServerProcess**.

See Also

CoRevokeClassObject, **CoReleaseServerProcess**, **Out-of-process Server Implementation Helpers**

CoTaskMemAlloc Quick Info

Allocates a block of task memory in the same way that **IMalloc::Alloc** does.

```
LPVOID CoTaskMemAlloc(
  ULONG cb      //Size in bytes of memory block to be allocated
);
```

Parameter

cb

[in] Size, in bytes, of the memory block to be allocated.

Return Values

Allocated memory block

Memory block allocated successfully.

NULL

Insufficient memory available.

Remarks

The **CoTaskMemAlloc** function uses the default allocator to allocate a memory block in the same way that **IMalloc::Alloc** does. It is not necessary to call the **CoGetMalloc** function before calling **CoTaskMemAlloc**.

The initial contents of the returned memory block are undefined - there is no guarantee that the block has been initialized. The allocated block may be larger than *cb* bytes because of the space required for alignment and for maintenance information.

If *cb* is zero, **CoTaskMemAlloc** allocates a zero-length item and returns a valid pointer to that item. If there is insufficient memory available, **CoTaskMemAlloc** returns NULL.

Note Applications should always check the return value from this method, even when requesting small amounts of memory, because there is no guarantee the memory will be allocated.

See Also

IMalloc::Alloc, **CoGetMalloc**, **CoTaskMemFree**, **CoTaskMemRealloc**

CoTaskMemFree Quick Info

Frees a block of task memory previously allocated through a call to the **CoTaskMemAlloc** or **CoTaskMemRealloc** function.

```
void CoTaskMemFree(

  void pv      //Pointer to memory block to be freed
);
```

Parameter

pv

[in] Pointer to the memory block to be freed.

Remarks

The **CoTaskMemFree** function, using the default OLE allocator, frees a block of memory previously allocated through a call to the **CoTaskMemAlloc** or **CoTaskMemRealloc** function.

The number of bytes freed equals the number of bytes that were originally allocated or reallocated. After the call, the memory block pointed to by *pv* is invalid and can no longer be used.

Note The *pv* parameter can be NULL, in which case this method has no effect.

See Also

CoTaskMemAlloc, **CoTaskMemRealloc**, **CoGetMalloc**, **IMalloc::Free**

CoTaskMemRealloc Quick Info

Changes the size of a previously allocated block of task memory.

```
LPVOID CoTaskMemRealloc(

  LPVOID pv,     //Pointer to memory block to be reallocated
  ULONG cb       //Size of block to be reallocated
);
```

Parameters

pv

[in] Pointer to the memory block to be reallocated. It can be a NULL pointer, as discussed in the Remarks.

cb

[in] Size, in bytes, of the memory block to be reallocated. It can be zero, as discussed in the following remarks.

Return Values

Reallocated memory block

Memory block successfully reallocated.

NULL

Insufficient memory or *cb* is zero and *pv* is not NULL.

Remarks

The **CoTaskMemRealloc** function changes the size of a previously allocated memory block in the same way that **IMalloc::Realloc** does. It is not necessary to call the **CoGetMalloc** function to get a pointer to the OLE allocator before calling **CoTaskMemRealloc**.

The *pv* argument points to the beginning of the memory block. If *pv* is NULL, **CoTaskMemRealloc** allocates a new memory block in the same way as the **CoTaskMemAlloc** function. If *pv* is not NULL, it should be a pointer returned by a prior call to **CoTaskMemAlloc**.

The *cb* argument specifies the size (in bytes) of the new block. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block can be in a different location. Because the new block can be in a different memory location, the pointer returned by **CoTaskMemRealloc** is not guaranteed to be the pointer passed through the *pv* argument. If *pv* is not NULL and *cb* is zero, then the memory pointed to by *pv* is freed.

CoTaskMemRealloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is NULL if the size is zero and the buffer argument is not NULL, or if there is not enough memory available to expand the block to the given size. In the first case, the original block is freed; in the second, the original block is unchanged.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

See Also

CoTaskMemAlloc, CoTaskMemFree, CoGetMalloc, IMalloc::Realloc

CoTreatAsClass Quick Info

Establishes or removes an emulation, in which objects of one class are treated as objects of a different class.

```
STDAPI CoTreatAsClass(

  REFCLSID clsidOld,      //CLSID for the original object to be emulated
  REFCLSID clsidNew       //CLSID for the new object that emulates the original
);
```

Parameters

clsidOld

[in] CLSID of the object to be emulated.

clsidNew

[in] CLSID of the object that should emulate the original object. This replaces any existing emulation for *clsidOld*. Can be CLSID_NULL, in which case any existing emulation for *clsidOld* is removed.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The emulation was successfully established or removed.

REGDB_E_CLASSNOTREG

The *clsidOld* parameter is not properly registered in the registration database.

REGDB_E_READREGDB

Error reading from registration database.

REGDB_E_WRITEREGDB

Error writing to registration database.

Remarks

This function sets the **TreatAs** entry in the registry for the specified object, allowing the object to be emulated by another application. Emulation allows an application to open and edit an object of a different format, while retaining the original format of the object. After this entry is set, whenever any function like **CoGetClassObject** specifies the object's original CLSID (*clsidOld*), it is transparently forwarded to the new CLSID (*clsidNew*), thus launching the application associated with the **TreatAs** CLSID. When the object is saved, it can be saved in its native format, which may result in loss of edits not supported by the original format.

You would call **CoTreatAsClass** in two situations if your application supports emulation:

- In response to an end-user request (through a conversion dialog box) that a specified object be treated as an object of a different class (an object created under one application be run under another application, while retaining the original format information).
- In a setup program, to register that one class of objects be treated as objects of a different class.

An example of the first case is that an end user might wish to edit a spreadsheet created by one application using a different application that can read and write the spreadsheet format of the original application. For an application that supports emulation, **CoTreatAsClass** can be called to implement a Treat As option in a conversion dialog box.

An example of the use of **CoTreatAsClass** in a setup program would be in an updated version of an application. When the application is updated, the objects created with the earlier version can be activated and treated as objects of the new version, while retaining the previous format information. This would allow you to give the user the option to convert when they save, or to save it in the previous format, possibly losing format information not available in the older version.

One result of setting an emulation is that when you enumerate verbs, as in the **IOleObject::EnumVerbs** method implementation in the default handler, this would enumerate the verbs from *clsidNew* instead of *clsidOld*.

To ensure that existing emulation information is removed when you install an application, your setup programs should call **CoTreatAsClass**, setting the *clsidNew* parameter to CLSID_NULL to remove any existing emulation for the classes they install.

If there is no CLSID assigned to the **AutoTreatAs** key in the registry, setting *clsidNew* and *clsidOld* to the same value removes the **TreatAs** entry, so there is no emulation. If there is a CLSID assigned to the **AutoTreatAs** key, that CLSID is assigned to the **TreatAs** key.

The **CoTreatAsClass** function does not validate whether an appropriate registry entry for *clsidNew* currently exists.

See Also

CoGetTreatAsClass

CoUninitialize Quick Info

Closes the OLE Component Object Model (COM) library, freeing any resources that it maintains and forcing all RPC connections to close.

void CoUninitialize();

Remarks

The **CoInitialize** and **CoUninitialize** calls must be balanced - if there are multiple calls to the **CoInitialize** function, there must be the same number of calls to **CoUninitialize**. Only the **CoUninitialize** call corresponding to the **CoInitialize** call that initialized the library can close it.

The **OleUninitialize** function calls **CoUninitialize** internally, so applications that call **OleUninitialize** do not also need to call **CoUninitialize**.

CoUninitialize should be called on application shutdown, as the last call made to the COM library after the application hides its main windows and falls through its main message loop. If there are open conversations remaining, **CoUninitialize** starts a modal message loop and dispatches any pending messages from the containers or server for this OLE application. By dispatching the messages, **CoUninitialize** ensures that the application does not quit before receiving all of its pending messages. Non-OLE messages are discarded.

See Also

CoInitialize, **OleUninitialize**

CoUnmarshalHresult Quick Info

Unmarshals an **HRESULT** type from the specified stream.

```
STDAPI CoUnmarshalHresult(

  LPSTREAM pStm,        //Pointer to stream used for unmarshaling
  HRESULT * phresult    //Pointer to the HRESULT
);
```

Parameters

pStm

[in] Pointer to the stream from which the **HRESULT** is to be unmarshaled.

phresult

[out] Pointer to the unmarshaled **HRESULT**.

Return Values

This function supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The **HRESULT** was unmarshaled successfully.

STG_E_INVALIDPOINTER

pStm is an invalid pointer.

Remarks

You do not explicitly call this function unless you are performing custom marshaling (that is, writing your own implementation of **IMarshal**), and your implementation needs to unmarshal an **HRESULT**.

You must use **CoUnmarshalHresult** to unmarshal **HRESULT**s previously marshaled by a call to the **CoMarshalHresult** function.

This function performs the following tasks:

1. Reads an HRESULT from a stream.
2. Returns the HRESULT.

See Also

CoMarshalHresult, **IStream**

CoUnmarshalInterface Quick Info

Initializes a newly created proxy using data written into the stream by a previous call to the **CoMarshalInterface** function, and returns an interface pointer to that proxy.

```
STDAPI CoUnmarshalInterface(
  IStream * pStm,      //Pointer to the stream
  REFIID riid,         //Reference to the identifier of the interface
  void ** ppv          //Indirect pointer to the unmarshaled interface
);
```

Parameters

pStm

[in] Pointer to the stream from which the interface is to be unmarshaled.

riid

[in] Reference to the identifier of the interface to be unmarshaled.

ppv

[out] Indirect pointer to the interface that was unmarshaled.

Return Values

This function supports the standard return value E_FAIL, as well as the following:

S_OK

The interface pointer was unmarshaled successfully.

STG_E_INVALIDPOINTER

pStm is an invalid pointer.

CO_E_NOTINITIALIZED

The **CoInitialize** or **OleInitialize** function was not called on the current thread before this function was called.

CO_E_OBJNOTCONNECTED

The object application has been disconnected from the remoting system (for example, as a result of a call to the **CoDisconnectObject** function).

REGDB_E_CLASSNOTREG

An error occurred reading the registration database.

E_NOINTERFACE

The final **QueryInterface** of this function for the requested interface returned E_NOINTERFACE.

CoCreateInstance errors

An error occurred when creating the handler.

Remarks

The **CoUnmarshalInterface** function performs the following tasks:

1. Reads from the stream the CLSID to be used to create an instance of the proxy.
2. Gets an **IMarshal** pointer to the proxy that is to do the unmarshaling. If the object uses COM's default marshaling implementation, the pointer thus obtained is to an instance of the generic proxy object. If the marshaling is occurring between two threads in the same process, the pointer is to an instance of the in-process free threaded marshaler. If the object provides its own marshaling code, **CoUnmarshalInterface** calls the **CoCreateInstance** function, passing the CLSID it read from the marshaling stream. **CoCreateInstance** creates an instance of the object's proxy and returns an **IMarshal** interface pointer to the proxy.
3. Using whichever **IMarshal** interface pointer it has acquired, the function then calls **IMarshal::UnmarshalInterface** and, if appropriate, **IMarshal::ReleaseMarshalData**.

The primary caller of this function is COM itself, from within interface proxies or stubs that unmarshal an interface pointer. There are, however, some situations in which you might call **CoUnmarshalInterface**. For example, if you are implementing a stub, your implementation would call **CoUnmarshalInterface** when the stub receives an interface pointer as a parameter in a method call.

See Also

CoMarshalInterface, **IMarshal::UnmarshalInterface**

CreateAntiMoniker Quick Info

Creates and supplies a new anti-moniker.

```
WINOLEAPI CreateAntiMoniker(

  LPMONIKER FAR *ppmk      //Indirect pointer to the anti-moniker
);
```

Parameter

ppmk

[out] Indirect pointer to the **IMoniker** interface on the new anti-moniker. When successful, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. When an error occurs, the pointer is NULL.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The anti-moniker has been created successfully.

Remarks

You would call this function only if you are writing your own moniker class (implementing the **IMoniker** interface). If you are writing a new moniker class that has no internal structure, you can use **CreateAntiMoniker** in your implementation of the **IMoniker::Inverse** method, and then check for an anti-moniker in your implementation of **IMoniker::ComposeWith**.

Like the ".." directory in MS-DOS file systems, which acts as the inverse to any directory name just preceding it in a path, an anti-moniker acts as the inverse of a simple moniker that precedes it in a composite moniker. An anti-moniker is used as the inverse of simple monikers with no internal structure. For example, the system-provided implementations of file monikers, item monikers, and pointer monikers all use anti-monikers as their inverse; consequently, an anti-moniker composed to the right of one of these monikers composes to nothing.

A moniker client (an object that is using a moniker to bind to another object) typically does not know the class of a given moniker, so the client cannot be sure that an anti-moniker is the inverse. Therefore, to get the inverse of a moniker, you would call **IMoniker::Inverse** rather than **CreateAntiMoniker**.

To remove the last piece of a composite moniker, you would do the following:

1. Call **IMoniker::Enum** on the composite, specifying FALSE as the first parameter. This creates an enumerator that returns the component monikers in reverse order.
2. Use the enumerator to retrieve the last piece of the composite.
3. Call **IMoniker::Inverse** on that moniker. The moniker returned by **IMoniker::Inverse** will remove the last piece of the composite.

See Also

IMoniker::Inverse, **IMoniker::ComposeWith**, **IMoniker - Anti-Moniker Implementation**

CreateBindCtx Quick Info

Supplies a pointer to an implementation of **IBindCtx** (a bind context object). This object stores information about a particular moniker-binding operation. The pointer this function supplies is required as a parameter in many methods of the **IMoniker** interface and in certain functions related to monikers.

```
WINOLEAPI CreateBindCtx(
  DWORD reserved,      //Reserved for future use
  LPBC FAR* ppbc       //Indirect pointer to the bind context
);
```

Parameters

reserved

[in] Reserved for future use; must be zero.

ppbc

[out] Indirect pointer to an **IBindCtx** interface on the new bind context object. When the function is successful, the caller is responsible for calling **IUnknown::Release** on the parameter. A NULL value indicates that an error occurred.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The bind context was allocated and initialized successfully.

Remarks

CreateBindCtx is most commonly used in the process of binding a moniker (locating and getting a pointer to an interface by identifying it through a moniker), as in the following steps:

1. Get a pointer to a bind context by calling the **CreateBindCtx** function.
2. Call the **IMoniker::BindToObject** method on the moniker, retrieving an interface pointer to the object to which the moniker refers.
3. Release the bind context.
4. Use the interface pointer.
5. Release the interface pointer.

The following code fragment illustrates these steps:

```
// pMnk is an IMoniker * that points to a previously acquired moniker
IFoo *pFoo;
IBindCtx *pbc;

CreateBindCtx( 0, &pbc );
pMnk->BindToObject( pbc, NULL, IID_IFoo, &pFoo );
pbc->Release();
```

```
// pFoo now points to the object; safe to use pFoo
pFoo->Release();
```

Bind contexts are also used in other methods of the **IMoniker** interface besides **IMoniker::BindToObject** and in the **MkParseDisplayName** function.

A bind context retains references to the objects that are bound during the binding operation, causing the bound objects to remain active (keeping the object's server running) until the bind context is released. Reusing a bind context when subsequent operations bind to the same object can improve performance. You should, however, release the bind context as soon as possible, because you could be keeping the objects activated unnecessarily.

A bind context contains a **BIND_OPTS** structure, which contains parameters that apply to all steps in a binding operation. When you create a bind context using **CreateBindCtx**, the fields of the **BIND_OPTS** structure are initialized to the following values:

```
cbStruct = sizeof(BIND_OPTS)
grfFlags = 0
grfMode = STGM_READWRITE
dwTickCountDeadline = 0.
```

You can call the **IBindCtx::SetBindOptions** method to modify these default values.

See Also

BIND_OPTS, **IBindCtx**, **IMoniker**, **MkParseDisplayName**

CreateClassMoniker

Creates a file moniker based on the specified path.

```
WINOLEAPI CreateClassMoniker(
  REFCLSID rclsid,          //Class this moniker binds to
  IMoniker **ppmk           //Indirect pointer to class moniker
);
```

Parameters

rclsid

[in] Reference to the CLSID of the object type to which this moniker binds.

ppmk

[out] When successful, indirect pointer to the **IMoniker** interface on the new class moniker. In this case, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. When an error occurs, the value of the pointer is NULL.

Return Values

S_OK

The moniker has been created successfully.

E_INVALIDARG

One or more arguments are invalid.

Remarks

CreateClassMoniker creates a class moniker that refers to the given class. The class moniker will supports binding to a fresh instance of the class identified by the CLSID in *rclsid*..

See Also

IMoniker - Class Moniker Implementation

CreateDataAdviseHolder Quick Info

Supplies a pointer to the OLE implementation of **IDataAdviseHolder** on the data advise holder object.

```
WINOLEAPI CreateDataAdviseHolder(

  IDataAdviseHolder **ppDAHolder      //Indirect pointer to the advise holder object
);
```

Parameter

ppDAHolder

[out] Indirect pointer to the **IDataAdviseHolder** interface on the new advise holder object.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The advise holder object has been instantiated and the pointer supplied.

Remarks

Call **CreateDataAdviseHolder** in your implementation of **IDataObject::DAdvise** to get a pointer to the OLE implementation of **IDataAdviseHolder** interface. With this pointer, you can then complete the implementation of **IDataObject::DAdvise** by calling the **IDataAdviseHolder::Advise** method, which creates an advisory connection between the calling object and the data object.

See Also

IDataAdviseHolder

CreateDataCache Quick Info

Supplies a pointer to a new instance of an OLE-provided implementation of a data cache.

```
WINOLEAPI CreateDataCache(
  LPUNKNOWN pUnkOuter,      //Pointer to whether cache is to be aggregated
  REFCLSID rclsid,          //CLSID used to generate icon labels
  REFIID riid,              //Reference to the identifier of the interface
  LPVOID FAR *ppvObj        //Indirect pointer to interface on supplied cache object
);
```

Parameters

pUnkOuter

[in] If the cache is to be created as part of an aggregate, pointer to the controlling **IUnknown** of the aggregate. If not, the parameter should be NULL.

rclsid

[in] CLSID used to generate icon labels. This value is typically CLSID_NULL.

riid

[in] Reference to the identifier of the interface the caller wants to use to communicate with the cache. This value is typically **IID_IOleCache** (defined in the OLE headers to equal the interface identifier for **IOleCache**).

ppvObj

[out] Iindirect points to the interface requested in *riid* on the cache object.

Return Values

This function supports the standard return values E_INVALIDARG and

E_OUTOFMEMORY, as well as the following:

S_OK

The OLE-provided cache was instantiated and the pointer supplied.

E_NOINTERFACE

The interface represented by *riid* is not supported by the object. The parameter *ppvObj* is set to NULL.

Remarks

The cache object created by **CreateDataCache** supports the **IOleCache**, **IOleCache2**, and **IOleCacheControl** interfaces for controlling the cache. It also supports the **IPersistStorage**, **IDataObject** (without advise sinks), **IViewObject**, and **IViewObject2** interfaces.

See Also

IOleCache, **IOleCache2**, **IOleCacheControl**

CreateFileMoniker Quick Info

Creates a file moniker based on the specified path.

```
WINOLEAPI CreateFileMoniker(
  LPCOLESTR lpszPathName,      //Pointer to path to be used
  LPMONIKER FAR *ppmk          //Indirect pointer to file moniker
);
```

Parameters

lpszPathName

[in] Pointer to a zero-terminated wide character string (two bytes per character) containing the path on which this moniker is based.

ppmk

[out] When successful, indirect pointer to the **IMoniker** interface on the new file moniker. In this case, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. When an error occurs, the value of the pointer is NULL.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The moniker has been created successfully.

MK_E_SYNTAX

Error in the syntax of a path was encountered while creating a moniker.

Remarks

CreateFileMoniker creates a moniker for an object that is stored in a file. A moniker provider (an object that provides monikers to other objects) can call this function to create a moniker to identify a file-based object that it controls, and can then make the pointer to this moniker available to other objects. An object identified by a file moniker must also implement the **IPersistFile** interface so it can be loaded when a file moniker is bound.

When each object resides in its own file, as in an OLE server application that supports linking only to file-based documents in their entirety, file monikers are the only type of moniker necessary. To identify objects smaller than a file, the moniker provider must use another type of moniker (such as an item moniker) in addition to file monikers, creating a composite moniker. Composite monikers would be needed in an OLE server application that supports linking to objects smaller than a document (such as sections of a document or embedded objects).

The *lpszPathName* can be a relative path, a UNC path (e.g., *server**share**path*), or a drive-letter-based path (e.g., c:\). If based on a relative path, the resulting moniker must be composed onto another file moniker before it can be bound.

A file moniker can be composed to the right only of another file moniker when the first moniker is based on an absolute path and the other is a relative path, resulting in a single file moniker based on the

combination of the two paths. A moniker composed to the right of another moniker must be a refinement of that moniker, and the file moniker represents the largest unit of storage. To identify objects stored within a file, you would compose other types of monikers (usually item monikers) to the right of a file moniker.

See Also

IMoniker - File Moniker Implementation

CreateGenericComposite Quick Info

Performs a generic composition of two monikers and supplies a pointer to the resulting composite moniker.

```
WINOLEAPI CreateGenericComposite(
  LPMONIKER pmkFirst,          //Pointer to the first moniker
  LPMONIKER pmkRest,           //Pointer to the second moniker
  LPMONIKER FAR *ppmkComposite //Indirect pointer to the composite
);
```

Parameters

pmkFirst

[in] Pointer to the moniker to be composed to the left of the moniker that *pmkRest* points to. Can point to any kind of moniker, including a generic composite.

pmkRest

[in] Pointer to the moniker to be composed to the right of the moniker that *pmkFirst* points to. Can point to any kind of moniker compatible with the type of the *pmkRest* moniker, including a generic composite.

ppmkComposite

[out] Indirect pointer to the **IMoniker** interface on the composite moniker object that is the result of composing *pmkFirst* and *pmkRest*. This object supports the OLE composite moniker implementation of **IMoniker**. When successful, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. If either *pmkFirst* or *pmkRest* are NULL, the supplied pointer is the one that is non-NULL. If both *pmkFirst* and *pmkRest* are NULL, or if an error occurs, the returned pointer is NULL.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The two input monikers were successfully composed.

MK_E_SYNTAX

The two monikers could not be composed due to an error in the syntax of a path (for example, if both *pmkFirst* and *pmkRest* are file monikers based on absolute paths).

Remarks

CreateGenericComposite joins two monikers into one. The moniker classes being joined can be different, subject only to the rules of composition. Call this function only if you are writing a new moniker class by implementing the **IMoniker** interface, within an implementation of **IMoniker::ComposeWith** that includes generic composition capability.

Moniker providers should call **IMoniker::ComposeWith** to compose two monikers together. Implementations of **ComposeWith** should (as do OLE implementations) attempt, when reasonable for the class, to perform non-generic compositions first, in which two monikers of the same class are combined. If

this is not possible, the implementation can call **CreateGenericComposite** to do a generic composition, which combines two monikers of different classes, within the rules of composition. You can define new types of non-generic compositions if you write a new moniker class.

During the process of composing the two monikers, **CreateGenericComposite** makes all possible simplifications. Consider the example where *pmkFirst* is the generic composite moniker, $A \bullet B \bullet C$, and *pmkRest* is the generic composite moniker, $C^{(-1)} \bullet B^{(-1)} \bullet Z$ (where $C^{(-1)}$ is the inverse of *C*). The function first composes *C* to $C^{(-1)}$, which composes to nothing. Then it composes *B* and $B^{(-1)}$ to nothing. Finally, it composes *A* to *Z*, and supplies a pointer to the generic composite moniker, $A \bullet Z$.

See Also

IMoniker::ComposeWith, **IMoniker - Generic Composite Moniker Implementation**

CreateILockBytesOnHGlobal Quick Info

Creates a byte array object that allows you use global memory as the physical device underneath a compound file implementation. This object supports an OLE implementation of the **ILockBytes** interface.

```
WINOLEAPI CreateILockBytesOnHGlobal(
  HGLOBAL hGlobal,          //Memory handle for the byte array object
  BOOL fDeleteOnRelease,    //Whether to free memory when the object is released
  ILockBytes ** ppLkbyt     //Indirect pointer to the new byte array object
);
```

Parameters

hGlobal

[in] Memory handle allocated by the **GlobalAlloc** function. The handle must be allocated as moveable and nondiscardable. If the handle is to be shared between processes, it must also be allocated as shared. New handles should be allocated with a size of zero. If *hGlobal* is NULL, **CreateILockBytesOnHGlobal** internally allocates a new shared memory block of size zero.

fDeleteOnRelease

[in] IWhether the underlying handle for this byte array object should be automatically freed when the object is released.

ppLkbyt

[out] Indirect pointer to the **ILockBytes** interface on the new byte array object.

Return Values

This function supports the standard return values E_INVALIDARG and

E_OUTOFMEMORY, as well as the following:

S_OK

The byte array object was created successfully.

Remarks

The **CreateILockBytesOnHGlobal** function creates a byte array object based on global memory. This object supports an OLE implementation of the **ILockBytes** interface, and is intended to be used as the basis for a compound file. You can then use the supplied **ILockBytes** pointer in a call to the **StgCreateDocfileOnILockBytes** function to build a compound file on top of this byte array object. The **ILockBytes** instance calls the **GlobalReAlloc** function to grow the memory block as needed.

The current contents of the memory block are undisturbed by the creation of the new byte array object. After creating the **ILockBytes** instance, you can use the **StgOpenStorageOnILockBytes** function to reopen a previously existing storage object already contained in the memory block. You can also call **GetHGlobalFromILockBytes** to get the global memory handle associated with the byte array object created by **CreateILockBytesOnHGlobal**.

Note If you free the *hGlobal* memory handle, the byte array object is no longer valid. You must call the **ILockBytes::Release** method before freeing the memory handle.

The value of the *hGlobal* parameter can be changed by a subsequent call to the **GlobalReAlloc** function; thus, you cannot rely on this value after the byte array object is created.

See Also

StgOpenStorageOnILockBytes, **GetHGlobalFromILockBytes**, **ILockBytes**

CreateItemMoniker Quick Info

Creates an item moniker that identifies an object within a containing object (typically a compound document).

```
WINOLEAPI CreateItemMoniker(
  LPCOLESTR lpszDelim,      //Pointer to delimiter string
  LPCOLESTR lpszItem,       //Pointer to item name
  LPMONIKER FAR *ppmk       //Indirect pointer to the item moniker
);
```

Parameters

lpszDelim

[in] Pointer to a wide character string (two bytes per character) zero-terminated string containing the delimiter (typically "!") used to separate this item's display name from the display name of its containing object.

lpszItem

[in] Pointer to a zero-terminated string indicating the containing object's name for the object being identified. This name can later be used to retrieve a pointer to the object in a call to **IOleItemContainer::GetObject**.

ppmk

[out] Indirect pointer to an **IMoniker** interface on the new item moniker. When successful, the function has called **IUnknown::AddRef** on the pointer and the caller is responsible for calling **IUnknown::Release**. If an error occurs, the supplied pointer has a NULL value.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The moniker was created successfully.

Remarks

A moniker provider, which hands out monikers to identify its objects so they are accessible to other parties, would call **CreateItemMoniker** to identify its objects with item monikers. Item monikers are based on a string, and identify objects that are contained within another object and can be individually identified using a string. The containing object must also implement the **IOleContainer** interface.

Most moniker providers are OLE applications that support linking. Applications that support linking to objects smaller than file-based documents, such as a server application that allows linking to a selection within a document, should use item monikers to identify the objects. Container applications that allow linking to embedded objects use item monikers to identify the embedded objects.

The *lpszItem* parameter is the name used by the document to uniquely identify the object. For example, if the object being identified is a cell range in a spreadsheet, an appropriate name might be something like "A1:E7." An appropriate name when the object being identified is an embedded object might be something like "embedobj1." The containing object must provide an implementation of the

IOleItemContainer interface that can interpret this name and locate the corresponding object. This allows the item moniker to be bound to the object it identifies.

Item monikers are not used in isolation. They must be composed with a moniker that identifies the containing object as well. For example, if the object being identified is a cell range contained in a file-based document, the item moniker identifying that object must be composed with the file moniker identifying that document, resulting in a composite moniker that is the equivalent of "C:\work\sales.xls!A1:E7."

Nested containers are allowed also, as in the case where an object is contained within an embedded object inside another document. The complete moniker of such an object would be the equivalent of "C:\work\report.doc!embedobj1!A1:E7." In this case, each containing object must call **CreateItemMoniker** and provide its own implementation of the **IOleItemContainer** interface.

See Also

IMoniker::ComposeWith, **IOleItemContainer**, **IMoniker - Item Moniker Implementation**

CreateOleAdviseHolder Quick Info

Creates an advise holder object for managing compound document notifications. It returns a pointer to the object's OLE implementation of the **IOleAdviseHolder** interface.

```
WINOLEAPI CreateOleAdviseHolder(

  ppOAHolder      //Indirect pointer to the advise holder object
);
```

Parameter

ppOAHolder

[out] Indirect pointer to the **IOleAdviseHolder** interface on the new advise holder object.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The new OLE advise holder returned successfully.

Remarks

The function **CreateOleAdviseHolder** creates an instance of an advise holder, which supports the OLE implementation of the **IOleAdviseHolder** interface. The methods of this interface are intended to be used to implement the advisory methods of **IOleObject**, and, when advisory connections have been set up with objects supporting an advisory sink, to send notifications of changes in the object to the advisory sink. The advise holder returned by **CreateOleAdviseHolder** will suffice for the great majority of applications. The OLE-provided implementation does not, however, support **IOleAdviseHolder::EnumAdvise,** so if you need to use this method, you will need to implement your own advise holder.

See Also

IOleAdviseHolder, **IOleObject**

CreatePointerMoniker Quick Info

Creates a pointer moniker based on a pointer to an object.

```
WINOLEAPI CreatePointerMoniker(

  LPUNKNOWN punk,          //Pointer to the interface to be used
  LPMONIKER FAR *ppmk      //Indirect pointer to the moniker
);
```

Parameters

punk

[in] Pointer to an **IUnknown** interface on the object to be identified by the resulting moniker.

ppmk

[out] Indirect pointer to the **IMoniker** interface on the new pointer moniker. When successful, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. When an error occurs, the returned pointer has a NULL value.

Return Values

This function supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The pointer moniker was created successfully.

Remarks

A pointer moniker wraps an existing interface pointer in a moniker that can be passed to those interfaces that require monikers. Pointer monikers allow an object that has no persistent representation to participate in a moniker-binding operation.

Pointer monikers are not commonly used, so this function is not often called.

See Also

IMoniker - Pointer Moniker Implementation

CreateStreamOnHGlobal Quick Info

Creates a stream object stored in global memory.

```
WINOLEAPI CreateStreamOnHGlobal(
  HGLOBAL hGlobal,          //Memory handle for the stream object
  BOOL fDeleteOnRelease,    //Whether to free memory when the object is released
  LPSTREAM * ppstm          //Indirect pointer to the new stream object
);
```

Parameters

hGlobal

[in] Memory handle allocated by the **GlobalAlloc** function. The handle must be allocated as moveable and nondiscardable. If the handle is to be shared between processes, it must also be allocated as shared. New handles should be allocated with a size of zero. If *hGlobal* is NULL, the **CreateStreamOnHGlobal** function internally allocates a new shared memory block of size zero.

fDeleteOnRelease

[in] Whether the underlying handle for this stream object should be automatically freed when the stream object is released.

ppstm

[out] Indirect pointer to the **IStream** interface on the new stream object. Its value cannot be NULL.

Return Values

This function supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The stream object was created successfully.

Remarks

The **CreateStreamOnHGlobal** function creates a stream object in memory that supports the OLE implementation of the **IStream** interface. The returned stream object supports both reading and writing, is not transacted, and does not support locking.

The initial contents of the stream are the current contents of the memory block provided in the *hGlobal* parameter. If the *hGlobal* paramter is NULL, this function internally allocates memory.

The current contents of the memory block are undisturbed by the creation of the new stream object. Thus, you can use this function to open an existing stream in memory.

The initial size of the stream is the size of the memory handle returned by the Win32 **GlobalSize** function. Because of rounding, this is not necessarily the same size that was originally allocated for the handle. If the logical size of the stream is important, you should follow the call to this function with a call to the **IStream::SetSize** method.

After you have created the stream object with **CreateStreamOnHGlobal**, you can call

GetHGlobalFromStream to get the global memory handle associated with the stream object.

See Also

CreateStreamOnHGlobal, **GetHGlobalFromStream**, **IStream::SetSize**, **GlobalSize** in Win32

DllCanUnloadNow Quick Info

Determines whether the DLL that implements this function is in use. If not, the caller can safely unload the DLL from memory.

Note OLE does not provide this function. DLLs that support the OLE Component Object Model (COM) should implement and export **DllCanUnloadNow**.

```
STDAPI DllCanUnloadNow();
```

Return Values

S_OK

The DLL can be unloaded.

S_FALSE

The DLL cannot be unloaded now.

Remarks

A call to **DllCanUnloadNow** determines whether the DLL from which it is exported is still in use. A DLL is no longer in use when it is not managing any existing objects (the reference count on all of its objects is 0).

Notes to Callers

You should not have to call **DllCanUnloadNow** directly. OLE calls it only through a call to the **CoFreeUnusedLibraries** function. When it returns S_OK, **CoFreeUnusedLibraries** safely frees the DLL.

Notes to Implementers

You need to implement **DllCanUnloadNow** in, and export it from, DLLs that are to be dynamically loaded through a call to the **CoGetClassObject** function. (You also need to implement and export the **DllGetClassObject** function in the same DLL).

If a DLL loaded through a call to **CoGetClassObject** fails to export **DllCanUnloadNow**, the DLL will not be unloaded until the application calls the **CoUninitialize** function to release the OLE libraries.

If the DLL links to another DLL, returning S_OK from **DllCanUnloadNow** will also cause the second, dependent DLL to be unloaded. To eliminate the possibility of a crash, the primary DLL should call the **CoLoadLibrary** function, specifying the path to the second DLL as the first parameter, and setting the auto free parameter to TRUE. This forces the COM library to reload the second DLL and set it up for a call to **CoFreeUnusedLibraries** to free it separately when appropriate.

DllCanUnloadNow should return S_FALSE if there are any existing references to objects that the DLL manages.

See Also

DllGetClassObject

DllGetClassObject Quick Info

Retrieves the class object from a DLL object handler or object application. **DllGetClassObject** is called from within the **CoGetClassObject** function when the class context is a DLL.

Note OLE does not provide this function. DLLs that support the OLE Component Object Model (COM) must implement **DllGetClassObject** in OLE object handlers or DLL applications.

```
STDAPI DllGetClassObject(
  REFCLSID rclsid,     //CLSID for the class object
  REFIID riid,         //Reference to the identifier of the interface that communicates with the class object
  LPVOID  * ppv        //Indirect pointer to the communicating interface
);
```

Parameters

rclsid

[in] CLSID that will associate the correct data and code.

riid

[in] Reference to the identifier of the interface that the caller is to use to communicate with the class object. Usually, this is **IID_IClassFactory** (defined in the OLE headers as the interface identifier for **IClassFactory**).

ppv

[out] Indirect pointer to the requested interface or, if an error occurs, to NULL.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The object was retrieved successfully.

CLASS_E_CLASSNOTAVAILABLE

The DLL does not support the class (object definition).

Remarks

If a call to the **CoGetClassObject** function finds the class object that is to be loaded in a DLL, **CoGetClassObject** uses the DLL's exported **DllGetClassObject** function.

Notes to Callers

You should not call **DllGetClassObject** directly. When an object is defined in a DLL, **CoGetClassObject** calls the **CoLoadLibrary** function to load the DLL, which, in turn, calls **DllGetClassObject**.

Notes to Implementers

You need to implement **DllGetClassObject** in (and export it from) DLLs that support the OLE Component Object Model.

Example

Following is an example (in C++) of an implementation of **DllGetClassObject**. In this example, **DllGetClassObject** creates a class object and calls its **QueryInterface** method to retrieve a pointer to the interface requested in *riid*. The implementation safely releases the reference it holds to the **IClassFactory** interface because it returns a reference-counted pointer to **IClassFactory** to the caller.

```
HRESULT_export  PASCAL DllGetClassObject
        (REFCLSID rclsid, REFIID riid, LPVOID * ppvObj)
{
    HRESULT hres = E_OUTOFMEMORY;
    *ppvObj = NULL;

    CClassFactory *pClassFactory = new CClassFactory(rclsid);
    if (pClassFactory != NULL)   {
        hRes = pClassFactory->QueryInterface(riid, ppvObj);
        pClassFactory->Release();
    }
    return hRes;
}
```

See Also

CoGetClassObject, **DllCanUnloadNow**

DllRegisterServer Quick Info

Instructs an in-process server to create its registry entries for all classes supported in this server module. If this function fails, the state of the registry for all its classes is indeterminate.

STDAPI DllRegisterServer(void);

Return Values

This function supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The registry entries were created successfully.

SELFREG_E_TYPELIB

The server was unable to complete the registration of all the type libraries used by its classes.

SELFREG_E_CLASS

The server was unable to complete the registration of all the object classes.

Remarks

E_NOTIMPL is not a valid return code.

See Also

DllUnregisterServer

DllUnregisterServer Quick Info

Instructs an in-process server to remove only those entries created through **DllRegisterServer**.

STDAPI DllUnregisterServer(void);

Return Values

This function supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The registry entries were created successfully.

S_FALSE

Unregistration of this server's known entries was successful, but other entries still exist for this server's classes.

SELFREG_E_TYPELIB

The server was unable to remove the entries of all the type libraries used by its classes.

SELFREG_E_CLASS

The server was unable to remove the entries of all the object classes.

Remarks

The server must not disturb any entries that it did not create which currently exist for its object classes. For example, between registration and unregistration, the user may have specified a **TreatAs** relationship between this class and another. In that case, unregistration can remove all entries except the **TreatAs** key and any others that were not explicitly created in **DllRegisterServer**. The Win32 registry functions specifically disallow the deletion of an entire populated tree in the registry. The server can attempt, as the last step, to remove the **CLSID** key, but if other entries still exist, the key will remain.

See Also

DllRegisterServer

DoDragDrop Quick Info

Carries out an OLE drag and drop operation.

```
WINOLEAPI DoDragDrop(
  IDataObject * pDataObject,     //Pointer to the data object
  IDropSource * pDropSource,     //Pointer to the source
  DWORD dwOKEffect,              //Effects allowed by the source
  DWORD * pdwEffect              //Pointer to effects on the source
);
```

Parameters

pDataObject

[in] Pointer to the **IDataObject** interface on a data object that contains the data being dragged.

pDropSource

[in] Pointer to an implementation of the **IDropSource** interface, which is used to communicate with the source during the drag operation.

dwOKEffect

[in] Effects the source allows in the OLE drag-and-drop operation. Most significant is whether it permits a move. The *dwOKEffect* and *pdwEffect* parameters obtain values from the **DROPEFFECT** enumeration. For a list of values, see **DROPEFFECT**.

pdwEffect

[out] Pointer to a value that indicates how the OLE drag-and-drop operation affected the source data. The *pdwEffect* parameter is set only if the operation is not canceled.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

DRAGDROP_S_DROP

The OLE drag-and-drop operation was successful.

DRAGDROP_S_CANCEL

The OLE drag-and-drop operation was canceled.

E_UNSPEC

Unexpected error occurred.

Remarks

If you are developing an application that can act as a data source for an OLE drag-and-drop operation, you must call **DoDragDrop** when you detect that the user has started an OLE drag-and-drop operation.

The **DoDragDrop** function enters a loop in which it calls various methods in the **IDropSource** and **IDropTarget** interfaces. (For a successful drag-and-drop operation, the application acting as the data source must also implement **IDropSource**, while the target application must implement **IDropTarget**.)

1. The **DoDragDrop** function determines the window under the current cursor location. It then checks to see if this window is a valid drop target.
2. If the window is a valid drop target, **DoDragDrop** calls **IDropTarget::DragEnter**. This method supplies an effect code indicating what would happen if the drop actually occurred. For a list of valid drop effects, see the **DROPEFFECT** enumeration.
3. **DoDragDrop** calls **IDropSource::GiveFeedback** with the effect code so that the drop source interface can provide appropriate visual feedback to the user. The *pDropSource* pointer passed into **DoDragDrop** specifies the appropriate **IDropSource** interface.
4. **DoDragDrop** tracks mouse cursor movements and changes in the keyboard or mouse button state.
   - If the user moves out of a window, **DoDragDrop** calls **IDropTarget::DragLeave**.
   - If the mouse enters another window, **DoDragDrop** determines if that window is a valid drop target and then calls **IDropTarget::DragEnter** for that window.
   - If the mouse moves but stays within the same window, **DoDragDrop** calls **IDropTarget::DragOver**.

5. If there is a change in the keyboard or mouse button state, **DoDragDrop** calls **IDropSource::QueryContinueDrag** and determines whether to continue the drag, to drop the data, or to cancel the operation based on the return value.
   - If the return value is S_OK, **DoDragDrop** first calls **IDropTarget::DragOver** to continue the operation. This method returns a new effect value and **DoDragDrop** then calls **IDropSource::GiveFeedback** with the new effect so appropriate visual feedback can be set. For a list of valid drop effects, see the **DROPEFFECT** enumeration. **IDropTarget::DragOver** and **IDropSource::GiveFeedback** are paired so that as the mouse moves across the drop target, the user is given the most up-to-date feedback on the mouse's position.
   - If the return value is DRAGDROP_S_DROP, **DoDragDrop** calls **IDropTarget::Drop**. The **DoDragDrop** function returns the last effect code to the source, so the source application can perform the appropriate operation on the source data, for example, cut the data if the operation was a move.
   - If the return value is DRAGDROP_S_CANCEL, the **DoDragDrop** function calls **IDropTarget::DragLeave**.

See Also

IDropSource

FACILITY_NT_BIT Quick Info

```
#define FACILITY_NT_BIT 0x10000000
```

Defines bits so macros are guaranteed to work.

FAILED Quick Info

```
#define FAILED(Status) ((HRESULT)(Status)<0)
```

Provides a generic test for failure on any status value. Negative numbers indicate failure.

FreePropVariantArray Quick Info

Calls **PropVariantClear** on each of the PROPVARIANTs in the *rgvar* array to zero the value of each of the members of the array.

```
HRESULT FreePropVariantArray(
    ULONG cVariant,          //Count of elements in the structure
    PROPVARIANT* rgvar[]     //Pointer to the PROPVARIANT
                             structure
);
```

Parameters

cVariant

[in] Count of elements in the **PROPVARIANT** array (*rgvar*).

rgvar

[in] Pointer to an initialized array of PROPVARIANT structures for which any deallocatable elements are to be freed. On exit, all zeroes are written to the PROPVARIANT (thus tagging them as VT_EMPTY).

Return Values

S_OK

The variant types are recognized and all items that can be freed have been freed.

STG_E_INVALID_PARAMETER

One or more PROPVARIANTs has an unknown type.

Remarks

FreePropVariantArray calls **PropVariantClear** on an array of PROPVARIANTs to clear all the valid members. All valid PROPVARIANTS are freed. If any of the PROPVARIANTs contain illegal VT-types, valid members are freed and the function returns STG_E_INVALIDPARAMETER.

Passing NULL for *rgvar* is legal, and produces a return code of S_OK.

See Also

PropVariantClear

GetClassFile Quick Info

Supplies the CLSID associated with the given filename.

```
WINOLEAPI GetClassFile(
  LPCWSTR szFileName,      //Pointer to filename for which you are requesting a CLSID
  CLSID * pclsid           //Pointer to location for returning the CLSID
);
```

Parameters

szFileName

[in] Points to the filename for which you are requesting the associated CLSID.

pclsid

[out] Points to the location where the associated CLSID is written on return.

Return Values

S_OK

Indicates the CLSID was successfully supplied.

MK_E_CANTOPENFILE

Indicates unable to open the specified filename.

MK_E_INVALIDEXTENSION

Indicates the specified extension in the registry is invalid.

Note This function can also return any file system errors.

Comments

When given a filename, the **GetClassFile** function finds the CLSID associated with that file. Examples of its use are in **OleCreateFromFile**, which is passed a file name and requires an associated CLSID, and in the OLE implementation of **IMoniker::BindToObject**, which, when a link to a file-based document is activated, calls **GetClassFile** to locate the object application that can open the file.

GetClassFile uses the following strategies to determine an appropriate CLSID:

1. If the file contains a storage object, as determined by a call to the **StgIsStorageFile** function, **GetClassFile** returns the CLSID that was written with the **IStorage::SetClass** method.
2. If the file is not a storage object, the **GetClassFile** function attempts to match various bits in the file against a pattern in the registry. A pattern in the registry can contain a series of entries of the form:

   ```
   regdb key = offset, cb, mask, value
   ```

   The value of the *offset* item is an offset from the beginning or end of the file and the *cb* item is a length in bytes. These two values represent a particular byte range in the file. (A negative value for

the *offset* item is interpreted from the end of the file). The *mask* value is a bit mask that is used to perform a logical AND operation with the byte range specified by *offset* and *cb*. The result of the logical AND operation is compared with the *value* item. If the *mask* is omitted, it is assumed to be all ones.

Each pattern in the registry is compared to the file in the order of the patterns in the database. The first pattern where each of the *value* items matches the result of the AND operation determines the CLSID of the file. For example, the pattern contained in the following entries of the registry requires that the first four bytes be AB CD 12 34 and that the last four bytes be FE FE FE FE:

```
HKEY_CLASSES_ROOT
   FileType
     {12345678-0000-0001-C000-000000000095}
       0 = 0, 4, FFFFFFFF, ABCD1234
       1 = -4, 4, , FEFEFEFE
```

If a file contains such a pattern, the CLSID {12345678-0000-0001-C000-000000000095} will be associated with this file.

3. If the above strategies fail, the **GetClassFile** function searches for the File Extension key in the registry that corresponds to the *.ext* portion of the filename. If the database entry contains a valid CLSID, this function returns that CLSID.
4. If all strategies fail, the function returns MK_E_INVALIDEXTENSION.

See Also

WriteClassStg

GetConvertStg

Quick Info

Returns the current value of the convert bit for the specified storage object.

```
WINOLEAPI GetConvertStg(
  IStorage * pStg          //Points to the IStorage interface on the storage object
);
```

Parameter

pStg

[in] **IStorage** pointer to the storage object from which the convert bit is to be retrieved.

Return Values

S_OK

Indicates the convert bit is set to TRUE.

S_FALSE

Indicates the convert bit is cleared (FALSE).

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

IStorage::OpenStream, **IStorage::OpenStorage**, and **IStream::Read** storage and stream access errors.

Comments

The **GetConvertStg** function is called by object servers that support the conversion of an object from one format to another. The server must be able to read the storage object using the format of its previous CLSID and write the object using the format of its new CLSID to support the object's conversion. For example, a spreadsheet created by one application can be converted to the format used by a different application.

The convert bit is set by a call to the **SetConvertStg** function. A container application can call this function on the request of an end user, or a setup program can call it when installing a new version of an application. An end user requests converting an object through the Convert To dialog box. When an object is converted, the new CLSID is permanently assigned to the object, so the object is subsequently associated with the new CLSID.

Then, when the object is activated, its server calls the **GetConvertStg** function to retrieve the value of the convert bit from the storage object. If the bit is set, the object's CLSID has been changed, and the server must read the old format and write the new format for the storage object.

After retrieving the bit value, the object application should clear the convert bit by calling the **SetConvertStg** function with its *fConvert* parameter set to FALSE.

See Also

SetConvertStg

GetHGlobalFromILockBytes Quick Info

Retrieves a global memory handle to a byte array object created using the **CreateILockBytesOnHGlobal** function.

```
WINOLEAPI GetHGlobalFromILockBytes(
  ILockBytes * pLkbyt,        //Points to the byte array object
  HGLOBAL * phglobal          //Points to the current memory handle for the specified byte array
);
```

Parameters

pLkbyt

[in] Points to the **ILockBytes** interface on the byte array object previously created by a call to the **CreateILockBytesOnHGlobal** function.

phglobal

[out] Points to the current memory handle used by the specified byte array object.

Return Values

S_OK

Indicates the handle was returned successfully.

E_INVALIDARG

Indicates invalid value specified for the *pLkbyt* parameter. It can also indicate that the byte array object passed in is not one created by the **CreateILockBytesOnHGlobal** function.

Comments

After a call to **CreateILockBytesOnHGlobal**, which creates a byte array object on global memory, **GetHGlobalFromILockBytes** retrieves a pointer to the handle of the global memory underlying the byte array object. The handle this function returns might be different from the original handle due to intervening calls to the **GlobalRealloc** function.

The contents of the returned memory handle can be written to a clean disk file, and then opened as a storage object using the **StgOpenStorage** function.

This function only works within the same process from which the byte array was created.

See Also

StgOpenStorage, **CreateILockBytesOnHGlobal**

GetHGlobalFromStream Quick Info

Retrieves the global memory handle to a stream that was created through a call to the **CreateStreamOnHGlobal** function.

```
WINOLEAPI GetHGlobalFromStream(
  IStream * pstm,          //Points to the stream object
  HGLOBAL * phglobal       //Points to the current memory handle for the specified stream
);
```

Parameters

pstm

[in] **IStream** pointer to the stream object previously created by a call to the **CreateStreamOnHGlobal** function.

phglobal

[out] Points to the current memory handle used by the specified stream object.

Return Values

S_OK

Indicates the handle was successfully returned.

E_INVALIDARG

Indicates invalid value specified for the *pstm* parameter. It can also indicate that the stream object passed in is not one created by a call to the **CreateStreamOnHGlobal** function.

Comments

The handle this function returns may be different from the original handle due to intervening **GlobalRealloc** calls.

This function can be called only from within the same process from which the byte array was created.

See Also

CreateStreamOnHGlobal

GlobalRealloc in Win32

GetRunningObjectTable Quick Info

Supplies a pointer to the **IRunningObjectTable** interface on the local Running Object Table (ROT).

```
WINOLEAPI GetRunningObjectTable(

  DWORD reserved,                    //Reserved
  LPRUNNINGOBJECTTABLE *pprot        //Indirect pointer
);
```

Parameters

reserved

[in] Reserved for future use; must be zero.

pprot

[out] Indirect pointer to the **IRunningObjectTable** interface on the local ROT. When the function is successful, the caller is responsible for calling **IUnknown::Release** on the pointer. If an error occurs, *pprot* is undefined.

Return Values

This function supports the standard return value E_UNEXPECTED, as well as the following:

S_OK

An **IRunningObjectTable** pointer was successfully returned.

Remarks

Each workstation has a local ROT that maintains a table of the objects that have been registered as running on that machine. This function returns an **IRunningObjectTable** interface pointer, which provides access to that table.

Moniker providers, which hand out monikers that identify objects so they are accessible to others, should call **GetRunningObjectTable**. Use the interface pointer returned by this function to register your objects when they begin running, to record the times that those objects are modified, and to revoke their registrations when they stop running. See the **IRunningObjectTable** interface for more information.

Compound-document link sources are the most common example of moniker providers. These include server applications that support linking to their documents (or portions of a document) and container applications that support linking to embeddings within their documents. Server applications that do not support linking can also use the ROT to cooperate with container applications that support linking to embeddings.

If you are implementing the **IMoniker** interface to write a new moniker class, and you need an interface pointer to the ROT, call **IBindCtx::GetRunningObjectTable** rather than the **GetRunningObjectTable** function. This allows future implementations of the **IBindCtx** interface to modify binding behavior.

See Also

IBindCtx::GetRunningObjectTable, **IMoniker**, **IRunningObjectTable**

GetScode Quick Info

```
#define GetScode(hr) ((SCODE) (hr))
```

Extracts the **SCODE** from an **HRESULT.**

This macro is obsolete and should not be used.

HRESULT_CODE Quick Info

```
#define HRESULT_CODE(hr) ((hr) & 0xFFFF)
```

Returns the error code part of the **HRESULT**.

HRESULT_FACILITY Quick Info

```
#define HRESULT_FACILITY(hr) (((hr) >> 16) & 0x1fff)
```

Returns the facility from the **HRESULT**.

HRESULT_FROM_NT Quick Info

```
#define HRESULT_FROM_NT(x) ((HRESULT) ((x) | FACILITY_NT_BIT))
```

Maps an NT status value into an **HRESULT**.

HRESULT_FROM_WIN32 Quick Info

```
#define HRESULT_FROM_WIN32(x) (x ? ((HRESULT) (((x) & 0x0000FFFF) | (FACILITY_WIN32 << 16)
    | 0x80000000)) : 0 )
```

Maps a WIN32 error value into an **HRESULT**. Note that this assumes WIN32 errors fall in the range of -32k to 32k.

HRESULT_SEVERITY Quick Info

```
#define HRESULT_SEVERITY(hr) (((hr) >> 31) & 0x1)
```

Returns the severity bit from the **HRESULT**.

IIDFromString Quick Info

Converts a string generated by the **StringFromIID** function back into the original interface identifier (IID).

```
WINOLEAPI IIDFromString(

  LPOLESTR lpsz,      //Pointer to the string representation of the IID
  LPIID lpiid         //Pointer to the requested IID on return
);
```

Parameters

lpsz

[in] Pointer to the string representation of the IID.

lpiid

[out] Pointer to the requested IID on return.

Return Values

This function supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The string was successfully converted.

Remarks

The function converts the interface identifier in a way that guarantees different interface identifiers will always be converted to different strings.

See Also

StringFromIID

IsAccelerator Quick Info

Determines whether the keystroke maps to an accelerator in the given accelerator table.

```
BOOL IsAccelerator(

  HACCEL hAccel,        //Handle to accelerator table
  INT cAccelEntries,    //Number of entries in the accelerator table
  LPMSG lpMsg,          //Pointer to the keystroke message to be translated
  WORD * lpwCmd         //Pointer to return the corresponding command identifier
);
```

Parameters

hAccel

[in] Handle to the accelerator table.

cAccelEntries

[in] Number of entries in the accelerator table.

lpMsg

[in] Pointer to the keystroke message to be translated.

lpwCmd

[out] Pointer to where to return the corresponding command identifier if there is an accelerator for the keystroke. It may be NULL.

Return Values

TRUE

The message is for the object application.

FALSE

The message is not for the object and should be forwarded to the container.

Remarks

While an object is active in-place, the object *always* has first chance to translate the keystrokes into accelerators. If the keystroke corresponds to one of its accelerators, the object must *not* call the **OleTranslateAccelerator** function – even if its call to the Windows **TranslateAccelerator** function fails. Failure to process keystrokes in this manner can lead to inconsistent behavior.

If the keystroke is not one of the object's accelerators, then the object must call **OleTranslateAccelerator** to let the container try its accelerator translation.

The object's server can call **IsAccelerator** to determine if the accelerator message belongs to it. Some servers do accelerator translation on their own and do not call **TranslateAccelerator**. Those applications will not call **IsAccelerator**, because they already have the information.

See Also

OleTranslateAccelerator, **TranslateAccelerator** in Win32

IS_ERROR Quick Info

```
#define IS_ERROR(Status) ((unsigned long)(Status) >> 31 == SEVERITY_ERROR)
```

Provides a generic test for errors on any status value.

IsEqualGUID Quick Info

Determines whether two GUIDs are equal.

```
BOOL IsEqualGUID(

  REFGUID rguid1,     //GUID to compare to rguid2
  REFGUID rguid2      //GUID to compare to rguid1
);
```

Parameters

rguid1

[in] GUID to compare to *rguid2*.

rguid2

[in] GUID to compare to *rguid1*.

Return Values

TRUE

The GUIDs are equal.

FALSE

The GUIDs are not equal.

Remarks

IsEqualGUID is used by the **IsEqualCLSID** and **IsEqualIID** functions.

See Also

IsEqualCLSID, **IsEqualIID**

IsEqualCLSID Quick Info

Determines whether two CLSIDs are equal.

```
BOOL IsEqualCLSID(

  REFCLSID rclsid1,     //CLSID to compare to rclsid2
  REFCLSID rclsid2      //CLSID to compare to rclsid1
);
```

Parameters

rclsid1

[in] CLSID to compare to *rclsid2*.

rclsid2

[in] CLSID to compare to *rclsid1*.

Return Values

TRUE

The CLSIDs are equal.

FALSE

The CLSIDs are not equal.

See Also

IsEqualGUID, **IsEqualIID**

IsEqualIID Quick Info

Determines whether two interface identifiers are equal.

```
BOOL IsEqualIID(

  REFGUID riid1,     //Interface identifier to compare to riid2
  REFGUID riid2      //Interface identifier to compare to riid1
);
```

Parameters

riid1

[in] Interface identifier to compare with *riid2*.

riid2

[in] Interface identifier to compare with *riid1*.

Return Values

TRUE

The interface identifiers are equal.

FALSE

The interface identifiers are not equal.

See Also

IsEqualGUID, **IsEqualCLSID**

IsValidIid

This function is obsolete.

IsValidInterface

This function is obsolete.

IsValidPtrIn

This function is obsolete.

IsValidPtrOut Quick Info

This function is obsolete.

MAKE_HRESULT Quick Info

```
#define MAKE_HRESULT(sev,fac,code) \HRESULT) (((unsigned long)(sev)<<31) | ((unsigned long)
    (fac)<<16) | ((unsigned long)(code))) )
```

Creates an **HRESULT** value from component pieces of the 32-bit value.

MAKE_SCODE Quick Info

```
#define MAKE_SCODE(sev,fac,code) \((SCODE) (((unsigned long)(sev)<<31) | ((unsigned long)
    (fac)<<16) | ((unsigned long)(code))) )
```

Returns an **SCODE** given an **HRESULT**.

MkParseDisplayName

Quick Info

Converts a string into a moniker that identifies the object named by the string. This is the inverse of the **IMoniker::GetDisplayName** operation, which retrieves the display name associated with a moniker.

```
WINOLEAPI MkParseDisplayName(
  LPBC pbc,                    //Pointer to the bind context object
  LPCOLESTR szUserName,        //Pointer to display name
  ULONG FAR *pchEaten,         //Pointer to the number of characters consumed
  LPMONIKER FAR *ppmk          //Indirect pointer to the moniker
);
```

Parameters

pbc

[in] Pointer to the **IBindCtx** interface on the bind context object to be used in this binding operation.

szUserName

[in] Pointer to a zero-terminated wide character string (two bytes per character) containing the display name to be parsed.

pchEaten

[out] Pointer to the number of characters of *szUserName* that were consumed. If the function is successful, **pchEaten* is the length of *szUserName*; otherwise, it is the number of characters successfully parsed.

ppmk

[out] Indirect pointer to the **IMoniker** implementation on the moniker that was built from *szUserName*. When successful, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. If an error occurs, the supplied pointer value is NULL.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The parse operation was successful and the moniker was created.

MK_E_SYNTAX

Error in the syntax of a file name or an error in the syntax of the resulting composite moniker.

This function can also return any of the error values returned by **IMoniker::BindToObject**, **IOleItemContainer::GetObject**, or **IParseDisplayName::ParseDisplayName**.

Remarks

The **MkParseDisplayName** function parses a human-readable name into a moniker that can be used to identify a link source. The resulting moniker can be a simple moniker (such as a file moniker), or it can be a generic composite made up of the component moniker pieces. For example, the following display name:

```
"c:\mydir\somefile!item 1"
```

could be parsed into the following generic composite moniker:

```
(FileMoniker based on "c:\mydir\somefile") • (ItemMoniker based on "item 1")
```

The most common use of **MkParseDisplayName** is in the implementation of the standard Links dialog box, which allows an end user to specify the source of a linked object by typing in a string. You may also need to call **MkParseDisplayName** if your application supports a macro language that permits remote references (reference to elements outside of the document).

Parsing a display name often requires activating the same objects that would be activated during a binding operation, so it can be just as expensive (in terms of performance) as binding. Objects that are bound during the parsing operation are cached in the bind context passed to the function. If you plan to bind the moniker returned by **MkParseDisplayName**, it is best to do so immediately after the function returns, using the same bind context, which removes the need to activate objects a second time.

MkParseDisplayName parses as much of the display name as it understands into a moniker. The function then calls **IMoniker::ParseDisplayName** on the newly created moniker, passing the remainder of the display name. The moniker returned by **IMoniker::ParseDisplayName** is composed onto the end of the existing moniker and, if any of the display name remains unparsed, **IMoniker::ParseDisplayName** is called on the result of the composition. This process is repeated until the entire display name has been parsed.

The **MkParseDisplayName** function attempts the following strategies to parse the beginning of the display name, using the first one that succeeds:

1. The function looks in the Running Object Table for file monikers corresponding to all prefixes of *szDisplayName* that consist solely of valid file name characters. This strategy can identify documents that are as yet unsaved.
2. The function checks the maximal prefix of *szDisplayName*, which consists solely of valid file name characters, to see if an OLE 1 document is registered by that name (this may require some DDE broadcasts). In this case, the returned moniker is an internal moniker provided by the OLE 1 compatibility layer of OLE 2.
3. The function consults the file system to check whether a prefix of *szDisplayName* matches an existing file. The file name can be drive-absolute, drive-relative, working-directory relative, or begin with an explicit network share name. This is the common case.
4. If the initial character of *szDisplayName* is an '@', the function finds the longest string immediately following it that conforms to the legal ProgID syntax. The function converts this string to a CLSID using the **CLSIDFromProgID** function. If the CLSID represents an OLE 2 class, the function loads the corresponding class object and asks for an **IParseDisplayName** interface pointer. The resulting **IParseDisplayName** interface is then given the whole string to parse, starting with the '@'. If the CLSID represents an OLE 1 class, then the function treats the string following the ProgID as an OLE1/DDE link designator having <*filename*>!<*item*> syntax.

See Also

IMoniker::ParseDisplayName, **IMoniker::GetDisplayName**, **IParseDisplayName**

MonikerCommonPrefixWith Quick Info

Creates a new moniker based on the common prefix that this moniker (the one comprising the data of this moniker object) shares with another moniker. This function is intended to be called only in implementations of **IMoniker::CommonPrefixWith**.

```
WINOLEAPI MonikerCommonPrefixWith(
  LPMONIKER pmkThis,              //Pointer to the first moniker being compared
  LPMONIKER pmkOther,             //Pointer to the second moniker being compared
  LPMONIKER FAR *ppmkCommon       //Indirect pointer to a moniker
);
```

Parameters

pmkThis

[in] Pointer to the **IMoniker** interface on one of the monikers for which a common prefix is sought; usually the moniker in which this call is used to implement **IMoniker::CommonPrefixWith**.

pmkOther

[in] Pointer to the **IMoniker** interface on the other moniker to compare with the first moniker.

ppmkCommon

[out] When successful, indirect pointer to an **IMoniker** interface on a moniker based on the common prefix of *pmkThis* and *pmkOther*. In this case, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. If an error occurs, the supplied pointer value is NULL if an error occurs.

Return Values

This function supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

A common prefix exists that is neither *pmkThis* nor *pmkOther*.

MK_S_HIM

The entire *pmkOther* moniker is a prefix of the *pmkThis* moniker.

MK_S_ME

The entire *pmkThis* moniker is a prefix of the *pmkOther* moniker.

MK_S_US

The *pmkThis* and *pmkOther* monikers are equal.

MK_E_NOPREFIX

The monikers have no common prefix.

MK_E_NOTBINDABLE

This function was called on a relative moniker. It is not meaningful to take the common prefix of relative monikers.

Remarks

Call **MonikerCommonPrefixWith** only in the implementation of **IMoniker::CommonPrefixWith** for a new moniker class.

Your implementation of **IMoniker::CommonPrefixWith** should first check whether the other moniker is of a type that you recognize and handle in a special way. If not, you should call **MonikerCommonPrefixWith**, passing itself as *pmkThis* and the other moniker as *pmkOther*. **MonikerCommonPrefixWith** correctly handles the cases where either moniker is a generic composite.

You should call this function only if *pmkThis* and *pmkOther* are both absolute monikers (where an absolute moniker is either a file moniker or a generic composite whose leftmost component is a file moniker, and where the file moniker represents an absolute path). Do not call this function on relative monikers.

See Also

IMoniker::CommonPrefixWith

MonikerRelativePathTo Quick Info

Provides a moniker that, when composed onto the end of the first specified moniker (or one with a similar structure), yields the second specified moniker. This function is intended for use only by **IMoniker::RelativePathTo** implementations.

```
WINOLEAPI MonikerRelativePathTo(
  LPMONIKER pmkSrc,              //Pointer to the source identified by the moniker
  LPMONIKER pmkDest,             //Pointer to the destination identified by the moniker
  LPMONIKER FAR * ppmkRelPath,   //Indirect pointer to the relative moniker
  BOOL dwReserved                //Reserved; must be non-zero
);
```

Parameters

pmkSrc

[in] Pointer to the **IMoniker** interface on the moniker that, when composed with the relative moniker to be created, produces *pmkDest*. This moniker identifies the "source" of the relative moniker to be created.

pmkDest

[in] Pointer to the **IMoniker** interface on the moniker to be expressed relative to *pmkSrc*. This moniker identifies the destination of the relative moniker to be created.

ppmkRelPath

[out] Indirect pointer to an **IMoniker** interface on the new relative moniker. When successful, the function has called **IUnknown::AddRef** on the parameter and the caller is responsible for calling **IUnknown::Release**. If an error occurs, the pointer value is NULL.

dwReserved

[in] Reserved; must be non-zero.

Return Values

This function supports the standard return value E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

A meaningful relative path has been returned.

MK_S_HIM

The only form of the relative path is the other moniker.

MK_E_NOTBINDABLE

Indicates that *pmkSrc* is a relative moniker, such as an item moniker, and must be composed with the moniker of its container before a relative path can be determined.

Remarks

Call **MonikerRelativePathTo** only in the implementation of **IMoniker::RelativePathTo** if you are

implementing a new moniker class.

Your implementation of **IMoniker::RelativePathTo** should first check whether the other moniker is of a type you recognize and handle in a special way. If not, you should call **MonikerRelativePathTo**, passing itself as *pmkThis* and the other moniker as *pmkOther*. **MonikerRelativePathTo** correctly handles the cases where either moniker is a generic composite.

You should call this function only if *pmkSrc* and *pmkDest* are both absolute monikers, where an absolute moniker is either a file moniker or a generic composite whose leftmost component is a file moniker, and where the file moniker represents an absolute path. Do not call this function on relative monikers.

See Also

IMoniker::RelativePathTo

OleBuildVersion Quick Info

This function is obsolete.

OleConvertIStorageToOLESTREAM Quick Info

Converts the specified storage object from OLE 2 structured storage to the OLE 1 storage model but does not include the presentation data. This is one of several compatibility functions.

```
WINOLEAPI OleConvertIStorageToOLESTREAM(
  IStorage * pStg,              //Pointer to the OLE 2 storage object to be converted
  LPOLESTREAM lpolestream       //Pointer to the stream where the OLE1 storage is written
);
```

Parameters

pStg

[in] Pointer to the **IStorage** interface on the storage object to be converted to an OLE 1 storage.

lpolestream

[out] Pointer to an OLE 1 stream structure where the persistent representation of the object is saved using the OLE 1 storage model.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The storage object was successfully converted and the **OLESTREAM** structure contains the persistent representation of an OLE 1 object.

CONVERT10_E_STG_NO_STD_STREAM

Object cannot be converted because its storage is missing a stream.

CONVERT10_S_NO_PRESENTATION

The specified storage object contains a Paintbrush object in DIB format and there is no presentation data in the **OLESTREAM**.

Remarks

This function converts an OLE 2 storage object to OLE 1 format. The **OLESTREAM** code implemented for OLE 1 must be available.

On entry, the stream pointed to by *lpolestm* should be created and positioned just as it would be for an **OleSaveToStream** call. On exit, the stream contains the persistent representation of the object using OLE 1 storage.

Note Paintbrush objects are dealt with differently from other objects because their native data is in DIB format. When Paintbrush objects are converted using **OleConvertIStorageToOLESTREAM**, no presentation data is added to the **OLESTREAM**. To include presentation data, use the **OleConvertIStorageToOLESTREAMEx** function instead.

See Also

CoIsOle1Class, **OleConvertIStorageToOLESTREAMEx**, **OleConvertOLESTREAMToIStorage**, **OleConvertOLESTREAMToIStorageEx**

OleConvertIStorageToOLESTREAMEx Quick Info

Converts the specified storage object from OLE 2 structured storage to the OLE 1 storage model, including the presentation data. This is one of several compatibility functions.

```
WINOLEAPI OleConvertIStorageToOLESTREAMEx(
  IStorage   * pStg,          //Pointer to the OLE 2 storage object to be converted
  CLIPFORMAT  cfFormat,       //Presentation data format
  LONG        lWidth,         //Width in HIMETRIC
  LONG        lHeight,        //Height in HIMETRIC
  DWORD       dwSize,         //Size of data
  STGMEDIUM   pmedium,        //Pointer to data
  LPOLESTREAM lpolestm        //Pointer to the stream where the OLE1 storage is written
);
```

Parameters

pStg

[in] Pointer to the **IStorage** interface on the storage object to be converted to an OLE 1 storage.

cfFormat

[in] Format of the presentation data. May be NULL, in which case the *lWidth, lHeight, dwSize,* and *pmedium* parameters are ignored.

lWidth

[in] Width of the object presentation data in HIMETRIC units.

lHeight

[in] Height of the object presentation data in HIMETRIC units.

dwSize

[in] Size of the data, in bytes, to be converted.

pmedium

[in] Pointer to the **STGMEDIUM** structure for the serialized data to be converted. See **STGMEDIUM** for more information.

lpolestm

[out] Pointer to a stream where the persistent representation of the object is saved using the OLE 1 storage model.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The conversion was completed successfully.

DV_E_STGMEDIUM

The *hGlobal* member of **STGMEDIUM** is NULL.

DV_E_TYMED

The *tymed* member of the **STGMEDIUM** structure is not TYMED_HGLOBAL or TYMED_ISTREAM.

Remarks

The **OleConvertIStorageToOLESTREAMEx** function converts an OLE 2 storage object to OLE 1 format. It differs from the **OleConvertIStorageToOLESTREAM** function in that the presentation data to be written to the OLE 1 storage is passed in.

Because **OleConvertIStorageToOLESTREAMEx** can specify which presentation data to convert, it can be used by applications that do not use OLE default caching resources but do use OLE's conversion resources.

The value of the tymed member of **STGMEDIUM** must be either TYMED_HGLOBAL or TYMED_ISTREAM; refer to the **TYMED** enumeration for more information. The medium is not released by **OleConvertIStorageToOLESTREAMEx**.

See Also

CoIsOle1Class, **OleConvertIStorageToOLESTREAM**, **OleConvertOLESTREAMToIStorage**, **OleConvertOLESTREAMToIStorageEx**, **STGMEDIUM** structure, **TYMED** enumeration

OleConvertOLESTREAMToIStorage Quick Info

Converts the specified object from the OLE 1 storage model to an OLE 2 structured storage object without specifying presentation data. This is one of several compatibility functions.

```
WINOLEAPI OleConvertOLESTREAMToIStorage(
  LPOLESTREAM lpolestream,          //Pointer to the stream where the OLE 1 storage is written
  IStorage * pstg,                  //Pointer to OLE 2 storage object
  const DVTARGETDEVICE * ptd        //Pointer to target device
);
```

Parameters

lpolestream

[in] Pointer to a stream that contains the persistent representation of the object in the OLE 1 storage format.

pstg

[out] Pointer to the **IStorage** interface on the OLE 2 structured storage object.

ptd

[in] Pointer to the DVTARGETDEVICE structure specifying the target device for which the OLE 1 object is rendered.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

The object was successfully converted.

CONVERT10_S_NO_PRESENTATION

The object either has no presentation data or uses native data for its presentation.

DV_E_DVTARGETDEVICE or DV_E_DVTARGETDEVICE_SIZE

Invalid value for *ptd*.

Remarks

This function converts an OLE 1 object to an OLE 2 structured storage object. You can use this function to update OLE 1 objects to OLE 2 objects when a new version of the object application supports OLE 2.

On entry, the *lpolestm* parameter should be created and positioned just as it would be for an **OleLoadFromStream** function call. On exit, the *lpolestm* parameter is positioned just as it would be on exit from an **OleLoadFromStream** function, and the *pstg* parameter contains the uncommitted persistent representation of the OLE 2 storage object.

For OLE 1 objects that use native data for their presentation, the **OleConvertOLESTREAMToIStorage** function returns CONVERT10_S_NO_PRESENTATION. On receiving this return value, callers should call **IOleObject::Update** to get the presentation data so it can be written to storage.

Applications that do not use OLE's default caching resources, but do use the conversion resources, can use an alternate function, **OleConvertOLESTREAMToIStorageEx**, which can specify which presentation data to convert. In the **OleConvertOLESTREAMToIStorageEx** function, the presentation data read from the **OLESTREAM** structure is passed out and the newly created OLE 2 storage object does not contain a presentation stream.

The following steps describe the conversion process using **OleConvertOLESTREAMToIStorage**:

1. Create a root **IStorage** object by calling the **StgCreateDocfile** function(..., &*pstg*).
2. Open the OLE 1 file (using **OpenFile** or another OLE 1 technique).
3. Using the OLE 1 procedure for reading files, read from the file until an OLE object is encountered.
4. Allocate an **IStorage** object from the root **IStorage** created in step 1:

```
pstg->lpVtbl->CreateStorage(...&pStgChild);
hRes = OleConvertIStorageToOLESTREAM(polestm, pStgChild);
hRes = OleLoad(pStgChild, &IID_IOleObject, pClientSite, ppvObj);
```

5. Repeat step 3 until the file is completely read.

See Also

ColsOle1Class, **OleConvertIStorageToOLESTREAM**, **OleConvertIStorageToOLESTREAMEx**, **OleConvertOLESTREAMToIStorageEx**,**DVTARGETDEVICE** structure, **STGMEDIUM** structure, **TYMED** enumeration

OleConvertOLESTREAMToIStorageEx Quick Info

Converts the specified object from the OLE 1 storage model to an OLE 2 structured storage object including presentation data. This is one of several compatibility functions.

```
WINOLEAPI OleConvertOLESTREAMToIStorageEx(
  LPOLESTREAM lpolestm,        //Pointer to the stream where the OLE1 storage is written
  IStorage * pstg,             //Pointer to OLE 2 storage object
  CLIPFORMAT * pcfFormat,      //Pointer to presentation data
  LONG * plWidth,              //Points to width value
  LONG * plHeight,             //Pointer to height value
  DWORD * pdwSize,             //Pointer to size
  STGMEDIUM pmedium            //Pointer to the structure
);
```

Parameters

lpolestm

[in] Pointer to the stream that contains the persistent representation of the object in the OLE 1 storage format.

pstg

[out] Pointer to the OLE 2 structured storage object.

pcfFormat

[out] Pointer to where the format of the presentation data is returned. May be NULL, indicating the absence of presentation data.

plWidth

[out] Pointer to where the width value (in HIMETRIC) of the presentation data is returned.

plHeight

[out] Pointer to where the height value (in HIMETRIC) of the presentation data is returned.

pdwSize

[out] Pointer to where the size in bytes of the converted data is returned.

pmedium

[out] Pointer to where the **STGMEDIUM** structure for the converted serialized data is returned.

Return Values

S_OK

The conversion was completed successfully.

DV_E_TYMED|

Value of the *tymed* member of **STGMEDIUM** is not TYMED_ISTREAM or TYMED_NULL.

Remarks

This function converts an OLE 1 object to an OLE 2 structured storage object. You can use this function to update OLE 1 objects to OLE 2 objects when a new version of the object application supports OLE 2.

This function differs from the **OleConvertOLESTREAMToIStorage** function in that the presentation data read from the **OLESTREAM** structure is passed out and the newly created OLE 2 storage object does not contain a presentation stream.

Since this function can specify which presentation data to convert, it can be used by applications that do not use OLE's default caching resources but do use the conversion resources.

The *tymed* member of **STGMEDIUM** can only be TYMED_NULL or TYMED_ISTREAM. If it is TYMED_NULL, the data will be returned in a global handle through the *hGlobal* member of **STGMEDIUM**, otherwise data will be written into the *pstm* member of this structure.

See Also

CoIsOle1Class, **OleConvertIStorageToOLESTREAM**, **OleConvertIStorageToOLESTREAMEx**, **OleConvertOLESTREAMToIStorage** , **STGMEDIUM** structure, **TYMED** enumeration

OleCreate Quick Info

Creates an embedded object identified by a CLSID. You use it typically to implement the menu item that allows the end user to insert a new object.

```
WINOLEAPI OleCreate(
  REFCLSID rclsid,                  //CLSID of embedded object to be created
  REFIID riid,                      //Reference to the identifier of the interface used to communicate with new object
  DWORD renderopt,                  //RENDEROPT value indicating cached capabilities
  FORMATETC * pFormatEtc,           //Pointer to a FORMATETC structure
  IOleClientSite * pClientSite,     //Pointer to request services from the container
  IStorage * pStg,                  //Pointer to storage for the object
  void ** ppvObject                 //Indirect pointer to new object
);
```

Parameters

rclsid

[in] CLSID of the embedded object that is to be created.

riid

[in] Reference to the identifier of the interface, usually **IID_IOleObject** (defined in the OLE headers as the interface identifier for **IOleObject**), through which the caller will communicate with the new object.

renderopt

[in] A value from the enumeration **OLERENDER**, indicating the locally cached drawing capabilities the newly created object is to have. The **OLERENDER** value chosen affects the possible values for the *pFormatEtc* parameter.

pFormatEtc

[in] Depending on which of the **OLERENDER** flags is used as the value of *renderopt*, pointer to one of the **FORMATETC** enumeration values. Refer to the **OLERENDER** enumeration for restrictions. This parameter, along with the *renderopt* parameter, specifies what the new object can cache initially.

pClientSite

[in] If you want **OleCreate** to call **IOleObject::SetClientSite**, pointer to the **IOleClientSite** interface on the container. The value may be NULL, in which case you must specifically call **IOleClientSite::SetClientSite** before attempting operations.

pStg

[in] Pointer to an instance of the **IStorage** interface on the storage object. This parameter may not be NULL.

ppvObject

[out] Upon successful return, indirect pointer to the interface requested in *riid* on the newly created object.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

Embedded object created successfully.

Remarks

The **OleCreate** function creates a new embedded object, and is typically called to implement the menu item Insert New Object**.** When **OleCreate** returns, the object it has created is blank (contains no data), unless *renderopt* is OLERENDER_DRAW or OLERENDER_FORMAT, and is loaded. Containers typically then call the **OleRun** function or **IOleObject::DoVerb** to show the object for initial editing.

The *rclsid* parameter specifies the CLSID of the requested object. CLSIDs of registered objects are stored in the system registry. When an application user selects Insert Object, a selection box allows the user to select the type of object desired from those in the registry. When **OleCreate** is used to implement the Insert Object menu item, the CLSID associated with the selected item is assigned to the *rclsid* parameter of **OleCreate**.

The *riid* parameter specifies the interface the client will use to communicate with the new object. Upon successful return, the *ppvObject* parameter holds a pointer to the requested interface.

The created object's cache contains information that allows a presentation of a contained object when the container is opened. Information about what should be cached is passed in the *renderopt* and *pFormatetc* values. When **OleCreate** returns, the created object's cache is not necessarily filled. Instead, the cache is filled the first time the object enters the running state. The caller can add additional cache control with a call to **IOleCache::Cache** after the return of **OleCreate** and before the object is run. If *renderopt* is OLERENDER_DRAW or OLERENDER_FORMAT, **OleCreate** requires that the object support the **IOleCache** interface. There is no such requirement for any other value of *renderopt*.

If *pClientSite* is non-NULL, **OleCreate** calls **IOleObject::SetClientSite** through the *pClientSite* pointer. **IOleClientSite** is the primary interface by which an object requests services from its container. If *pClientSite* is NULL, you must make a specific call to **IOleObject::SetClientSite** before attempting any operations.

See Also

OLERENDER, **FORMATETC**, **IOleClientSite**, **IOleObject**

OleCreateDefaultHandler Quick Info

Creates a new instance of the default embedding handler. This instance is initialized so it creates a local server when the embedded object enters the running state.

```
WINOLEAPI OleCreateDefaultHandler(
  REFCLSID clsid,              //OLE server to be loaded
  LPUNKNOWN pUnkOuter,         //Pointer to controlling IUnknown if aggregated; else NULL
  REFIID riid,                 //Reference to the identifier of the interface for communicating with handler
  LPVOID FAR * ppvObj          //Indirect pointer to interface on handler
);
```

Parameters

clsid

[in] CLSID identifying the OLE server to be loaded when the embedded object enters the running state.

pUnkOuter

[in] Pointer to the controlling **IUnknown** interface if the handler is to be aggregated; NULL if it is not to be aggregated.

riid

[in] Reference to the identifier of the interface, usually **IID_IOleObject**, through which the caller will communicate with the handler.

ppvObj

[out] Indirect pointer to the interface requested in *riid* on the newly created handler.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

NOERROR

The creation operation was successful.

Remarks

OleCreateDefaultHandler creates a new instance of the default embedding handler, initialized so it creates a local server identified by the *clsid* parameter when the embedded object enters the running state. If you are writing a handler and want to use the services of the default handler, call **OleCreateDefaultHandler**. OLE also calls it internally when the CLSID specified in an object creation call is not registered.

If the given class does not have a special handler, a call to **OleCreateDefaultHandler** produces the same results as a call to the **CoCreateInstance** function with the class context parameter assigned the value CLSCTX_INPROC_HANDLER.

See Also

CoCreateInstance, **CLSCTX**

OleCreateEmbeddingHelper Quick Info

Creates an OLE embedding helper object using application-supplied code aggregated with pieces of the OLE default object handler. This helper object can be created and used in a specific context and role, as determined by the caller.

```
WINOLEAPI OleCreateEmbeddingHelper(
  REFCLSID clsid,              //Identifier of the class to be helped
  LPUNKNOWN pUnkOuter,         //Pointer to controlling IUnknown if aggregated; else NULL
  DWORD flags,                 //Purpose for the helper
  LPCLASSFACTORY pCF,          //Pointer on the class object for the secondary object
  REFIID riid,                 //Reference to the identifier of the interface desired by the caller
  LPVOID * ppvObj              //Indirect pointer to requested interface on helper
);
```

Parameters

clsid

[in] CLSID of the class to be helped.

pUnkOuter

[in] If the embedding helper is to be aggregated, pointer to the outer object's controlling **IUnknown** interface. If it is not to be aggregated, although this is rare, the value should be NULL.

flags

[in] DWORD containing flags that specify the role and creation context for the embedding helper. For legal values, see the following Remarks section.

pCF

[in] Pointer to the **IClassFactory** interface on the class object the function uses to create the secondary object. In some situations, this value may be NULL. For more information, see the following Remarks section.

riid

[in] Reference to the identifier of the interface desired by the caller.

ppvObj

[out] Indirect pointer to the requested interface on the newly created embedding helper.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The OLE embedding helper was created successfully.

E_NOINTERFACE

The interface is not supported by the object.

Remarks

The **OleCreateEmbeddingHelper** function creates an object that supports the same interface implementations found in the default handler, but which has additional hooks that allow it to be used more generally than just as a handler object. The following two calls produce the same result:

```
OleCreateEmbeddingHelper(clsid, pUnkOuter, EMBDHLP_INPROC_HANDLER |
    EMBDHLP_CREATENOW, NULL, iid, ppvObj)

OleCreateDefaultHandler(clsid, pUnkOuter, iid, ppvObj)
```

The embedding helper is aggregatable; *pUnkOuter* is the controlling **IUnknown** of the aggregate of which the embedding helper is to be a part. It is used to create a new instance of the OLE default handler, which can be used to support objects in various roles. The caller passes a pointer to its **IClassFactory** implementation to **OleCreateEmbeddingHelper**. This object and the default handler are then aggregated to create the new embedding helper object.

The **OleCreateEmbeddingHelper** function is usually used to support one of the following implementations:

- An EXE object application that is being used as both a container and a server, and which supports inserting objects into itself. For this case, **CreateEmbeddingHelper** allows the object to support the interfaces usually supported only in the handler. To accomplish this, the application must first register its CLSID for different contexts, making two registration calls to the **CoRegisterClassObject** function, rather than one, as follows:

  ```
  CoRegisterClassObject(clsidMe, pUnkCfLocal, CLSCTX_LOCAL_SERVER,
      REGCLS_MULTI_SEPARATE...)

  CoRegisterClassObject(clsidMe, pUnkCfInProc, CLSCTX_INPROC_SERVER,
      REGCLS_MULTI_SEPARATE...)
  ```

  In these calls, you would pass along different class factory implementations to each of *pUnkCfLocal* and *pUnkCfInProc*. The class factory pointed to by *pUnkCfLocal* would be used to create objects that are to be embedded in a remote process, which is the normal case which uses a handler object associated with the client. However, when a server both creates an object and embeds it within itself, *pUnkCfInProc* points to a class object that can create an object that supports the handler interfaces. The local class is used to create the object and the in-process class creates the embedding helper, passing in the pointer to the first object's class factory in *pCF*.
- A custom in-process object handler, in which case, the DLL creates the embedding helper by passing in a pointer to a private implementation of **IClassFactory** in *pCF*.

The *flags* parameter indicates how the embedding helper is to be used and how and when the embedding helper is initialized. The values for *flags* are obtained by OR-ing together values from the following table:

Values for *flags* Parameter	**Purpose**
EMBDHLP_INPROC_HANDLER	Creates an embedding helper that can be used with DLL object applications; specifically, the helper exposes the caching features of the default object handler.
EMBDHLP_INPROC_SERVER	Creates an embedding helper that is to be used as part of an in-process server. *pCF* cannot be NULL.
EMBDHLP_CREATENOW	Creates the secondary object using

	pCF immediately; if *pCF* is null, the standard proxy manager is used.
EMBDHLP_DELAYCREATE	Delays creation of the secondary object until it is needed (when the helper is put into the running state) to enhance speed and memory use. *pCF* must not be NULL. The EMBDHLP_INPROC_HANDLER flag cannot be used with this flag.

See Also

OleCreateDefaultHandler

OleCreateEx Quick Info

Extends **OleCreate** functionality by supporting more efficient instantiation of objects in containers requiring caching of multiple presentation formats or data, instead of the single format supported by **OleCreate**.

```
HRESULT OleCreateEx(
  REFCLSID rclsid,                  //Class of object to create
  REFIID riid,                      //Reference to the identifier of the interface of object to return
  DWORD dwFlags,                    //Value from OLECREATE enumeration
  DWORD renderopt,                  //Value from OLERENDER enumeration
  ULONG cFormats,                   //Number of FORMATETC structures in rgFormatEtc
  DWORD rgAdvf,                     //Points to an array of cFormats DWORD elements
  LPFORMATETC rgFormatEtc,          //Points to an array of cFormats FORMATETC structures; NULL otherwise
  LPADVISESINK pAdviseSink,         //IAdviseSink pointer (custom caching); NULL (default caching); NULL otherwise
  DWORD* rgdwConnection,            //Location to return array of dwConnection values
  LPOLECLIENTSITE pClientSite,      //Pointer to the primary interface the object will use to request services
  LPSTORAGE pStg,                   //Pointer to storage to use for object
  LPVOID FAR* ppvObj                //Indirect pointer to location to return riid interface
);
```

Parameters

rclsid

Identifies the class of the object to create.

riid

Reference to the identifier of the interface of the object to return.

dwFlags

Value taken from the **OLECREATE** enumeration.

renderopt

Value taken from the **RENDEROPT** enumeration.

cFormats

When *renderopt* is OLERENDER_FORMAT, indicates the number of **FORMATETC** structures in the *rgFormatEtc* array, which must be at least one. In all other cases, this parameter must be zero.

rgAdvf

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* DWORD elements, each of which is a combination of values from the **ADVF** enumeration. Each element of this array is passed in as the *advf* parameter to a call to either **IOleCache::Cache** or **IDataObject::DAdvise**, depending on whether *pAdviseSink* is NULL or non-NULL (see below). In all other cases, this parameter must be NULL.

rgFormatEtc

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **FORMATETC** structures. When *pAdviseSink* is NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to the object's **IOleCache::Cache**. This populates the data and presentation cache managed by the objects in-process handler (typically the default handler) with presentation or other cacheable

data. When *pAdviseSink* is non-NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to **IDataObject::DAdvise**. This allows the caller (typically an OLE Container) to do its own caching or processing of data received from the object. In all other cases, this parameter must be NULL.

pAdviseSink

When *renderopt* is OLERENDER_FORMAT, may be either a valid **IAdviseSink** pointer, indicating custom caching or processing of data advises, or NULL, indicating default caching of data formats. In all other cases, this parameter must be NULL.

rgdwConnection

Location to return the array of *dwConnection* values returned when the *pAdviseSink* interface is registered for each advisory connection using **IDataObject::DAdvise**, or NULL if the returned advisory connections are not needed. Must be NULL, if *pAdviseSink* is NULL.

pClientSite

Pointer to the primary interface through which the object will request services from its container. This parameter may be NULL, in which case it is the caller's responsibility to establish the client site as soon as possible using **IOleObject::SetClientSite**.

pStg

Pointer to the storage to use for the object and any default data or presentation caching established for it. This parameter may not be NULL.

ppvObj

Location to return the *riid* interface of the newly created object.

Return Values

This function supports the standard return value E_INVALIDARG, as well as the following:

S_OK

Success.

E_NOINTERFACE

The object does not support the *riid* interface.

Remarks

The following call to **OleCreate**:

```
    OleCreate(rclsid, riid, renderopt, pFormatEtc, pClientSite, pStg,
ppvObj);
```

is equivalent to the following call to **OleCreateEx**:

```
    DWORD    advf = ADVF_PRIMEFIRST;
    OleCreateEx(rclsid, riid, renderopt, 1, &advf, pFormatEtc, NULL,
pClientSite, pStg, ppvObj);
```

Existing instantiation functions, (**OleCreate**, **OleCreateFromFile**, **OleCreateFromData**, **OleCreateLink**, **OleCreateLinkToFile**, and **OleCreateLinkFromData**) create only a single presentation or data format cache in the default cache location (within the '\001OlePresXXX' streams of the passed-in **IStorage**), during instantiation. Plus, these caches must be created when the object next enters the running state. Since most applications require caching at least two presentations (screen and printer) and may require caching data in a different format or location from the handler, applications must typically launch and shut down the object server multiple times in order to prime their data caches during object creation, i.e., Insert

Object, Insert Object from File, and Paste Object.

Extended versions of these creation functions solve this problem. **OleCreateEx**, **OleCreateFromFileEx**, **OleCreateFromDataEx**, **OleCreateLinkEx**, **OleCreateLinkToFileEx**, and **OleCreateLinkFromDataEx** contain the following new parameters: *dwFlags* to indicate additional options, *cFormats* to indicate how many formats to cache, *rgAdvf*, from the **ADVF** enumeration, to specify the advise flags for each format to be cached, *pAdviseSink* to indicate whether presentation (default-handler) or data (non-default-handler) caching is required, *rgdwConnection* to return **IDataObject::DAdvise** cookies, and *pFormatEtc*, an array of formats rather than a single format.

Containers requiring that multiple presentations be cached on their behalf by the object's handler can simply call these functions and specify the number of formats in *cFormats*, the **ADVF** flags for each format in *rgAdvf*, and the set of formats in *pFormatEtc*. These containers pass NULL for *pAdviseSink*.

Containers performing all their own data- or presentation-caching perform these same steps, but pass a non-NULL *pAdviseSink*. They perform their own caching or manipulation of the object or data during **IAdviseSink::OnDataChange**. Typically such containers never establish the advisory connections with ADVF_NODATA, although they are not prevented from doing so.

These new functions are for OLE Compound Documents. Using these functions, applications can avoid the repeated launching and initialization steps required by the current functions. They are targeted at OLE Compound Document container applications that use default data- and presentation-caching, and also at applications that provide their own caching and data transfer from the underlying **IDataObject::DAdvise** support.

See Also

OleCreate, **IOleCache::Cache**, **IDataObject::DAdvise**, **IOleObject::SetClientSite**, **IAdviseSink::OnDataChange**, **IStorage**, **OLERENDER**, **FORMATETC**, **ADVF**

OleCreateFontIndirect Quick Info

Creates and initializes a standard font object using an initial description of the font's properties in a **FONTDESC** structure. The function returns an interface pointer to the new font object specified by caller in the *riid* parameter. A **QueryInterface** is built into this call. The caller is responsible for calling **Release** through the interface pointer returned.

```
STDAPI OleCreateFontIndirect(
  FONTDESC* pFontDesc,      //Pointer to the structure of parameters for font
  REFIID  riid,             //Reference to the identifier of the interface
  VOID** ppvObj             //Indirect pointer to the object
);
```

Parameters

pFontDesc

[in] Pointer to a caller-allocated structure containing the initial state of the font.

riid

[in] Reference to the identifier of the interface describing the type of interface pointer to return in *ppvObj*.

ppvObj

[out] Indirect pointer to the initial interface pointer on the new object. If successful, the caller is responsible to call **Release** through this interface pointer when the new object is no longer needed. If unsuccessful, the value of *ppvObj* is set to NULL.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The new font was created successfully.

E_NOINTERFACE

The object does not support the interface specified in *riid*.

E_POINTER

The address in *pFontDesc* or *ppvObj* is not valid. For example, it may be NULL.

See Also

FONTDESC

OleCreateFromData Quick Info

Creates an embedded object from a data transfer object retrieved either from the clipboard or as part of an OLE drag-and-drop operation. It is intended to be used to implement a paste from an OLE drag-and-drop operation.

```
WINOLEAPI OleCreateFromData(
  LPDATAOBJECT pSrcDataObj,          //Pointer to the data transfer object
  REFIID riid,                       //Reference to the identifier of the interface to be used to communicate with the nev
  DWORD renderopt,                   //Value from OLERENDER
  LPFORMATETC pFormatEtc,            //Pointer to value from FORMATETC, depending on renderopt
  LPOLECLIENTSITE pClientSite,       //Pointer to interface
  LPSTORAGE pStg,                    //Pointer to storage of object
  LPVOID FAR* ppvObj                 //Indirect pointer to the interface requested in riid
);
```

Parameters

pSrcDataObj

[in] Pointer to the **IDataObject** interface on the data transfer object that holds the data from which the object is created.

riid

[in] Reference to the identifier of the interface the caller later uses to communicate with the new object (usually **IID_IOleObject**, defined in the OLE headers as the interface identifier for **IOleObject**).

renderopt

[in] Value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-etrieval capabilities the newly created object is to have. Additional considerations are described in the following Remarks section.

pFormatEtc

[in] Pointer to a value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. The **OLERENDER** value chosen affects the possible values for the *pFormatEtc* parameter.

pClientSite

[in] Pointer to an instance of **IOleClientSite**, the primary interface through which the object will request services from its container. May be NULL.

pStg

[in] Pointer to the **IStorage** interface on the storage object. This parameter may not be NULL.

ppvObj

[out] Indirect pointer to the interface requested in *riid* on the newly created object.

Return Values

S_OK

The embedded object was created successfully.

OLE_E_STATIC

Indicates OLE can create only a static object.

DV_E_FORMATETC

No acceptable formats are available for object creation.

Remarks

The **OleCreateFromData** function creates an embedded object from a data transfer object supporting the **IDataObject** interface. The data object in this case is either the type retrieved from the clipboard with a call to the **OleGetClipboard** function or is part of an OLE drag-and-drop operation (the data object is passed to a call to **IDropTarget::Drop**).

If either the FileName or FileNameW clipboard format (CF_FILENAME) is present in the data transfer object, and CF_EMBEDDEDOBJECT or CF_EMBEDSOURCE do not exist, **OleCreateFromData** first attempts to create a package containing the indicated file. Generally, it takes the first available format. The Microsoft Windows NT File Manager places these formats on the clipboard when the user selects the File/Copy To Clipboard menu command.

If **OleCreateFromData** cannot create a package, it tries to create an object using the CF_EMBEDDEDOBJECT format. If that format is not available, **OleCreateFromData** tries to create it with the CF_EMBEDSOURCE format. If neither of these formats is available and the data transfer object supports the **IPersistStorage** interface, **OleCreateFromData** calls the object's **IPersistStorage::Save** to have the object save itself.

If an existing linked object is selected, then copied, it appears on the clipboard as just another embeddable object. Consequently, a paste operation that invokes **OleCreateFromData** may create a linked object. After the paste operation, the container should call the **QueryInterface** function, requesting **IID_IOleLink** (defined in the OLE headers as the interface identifier for **IOleLink**), to determine if a linked object was created.

Use the *renderopt* and *pFormatetc* parameters to control the caching capability of the newly created object. For general information about using the interaction of these parameters to determine what is to be cached, refer to the **OLERENDER** enumeration. There are, however, some additional specific effects of these parameters on the way **OleCreateFromData** initializes the cache.

When **OleCreateFromData** uses either the CF_EMBEDDEDOBJECT or the CF_EMBEDSOURCE clipboard format to create the embedded object, the main difference between the two is where the cache-initialization data is stored:

- CF_EMBEDDEDOBJECT indicates that the source is an existing embedded object. It already has in its cache the appropriate data, and OLE uses this data to initialize the cache of the new object.
- CF_EMBEDSOURCE indicates that the source data object contains the cache-initialization information in formats other than CF_EMBEDSOURCE. **OleCreateFromData** uses these to initialize the cache of the newly embedded object.

The *renderopt* values affect cache initialization as follows:

Value	Description
OLERENDER_DRAW & OLERENDER_FORMAT	If the presentation information to be cached is currently present in the appropriate cache-initialization pool, it is used. (Appropriate locations are in the source data object cache for

	CF_EMBEDDEDOBJECT, and in the other formats in the source data object for CF_EMBEDSOURCE.) If the information is not present, the cache is initially empty, but will be filled the first time the object is run. No other formats are cached in the newly created object.
OLERENDER_NONE	Nothing is to be cached in the newly created object. If the source has the CF_EMBEDDEDOBJECT format, any existing cached data that has been copied is removed.
OLERENDER_ASIS	If the source has the CF_EMBEDDEDOBJECT format, the cache of the new object is to contain the same cache data as the source object. For CF_EMBEDSOURCE, nothing is to be cached in the newly created object. This option should be used by more sophisticated containers. After this call, such containers would call **IOleCache::Cache** and **IOleCache::Uncache** to set up exactly what is to be cached. For CF_EMBEDSOURCE, they would then also call **IOleCache::InitCache**.

See Also

OleCreate, IDataObject

OleCreateFromDataEx Quick Info

Extends **OleCreateFromData** functionality by supporting more efficient instantiation of objects in containers requiring caching of multiple formats of presentation or data, instead of the single format supported by **OleCreateFromData**.

```
HRESULT OleCreateFromDataEx(
  LPDATAOBJECT pSrcDataObj,          //Pointer to data transfer object
  REFIID riid,                       //Reference to the identifier of the interface of object to return
  DWORD dwFlags,                     //Value from OLECREATE enumeration
  DWORD renderopt,                   //Value from OLERENDER enumeration
  ULONG cFormats,                    //Number of FORMATETCs in rgFormatEtc
  DWORD rgAdvf,                      //Points to an array of cFormats DWORD elements
  LPFORMATETC rgFormatEtc,           //Points to an array of cFormats FORMATETC structures
  LPADVISESINK pAdviseSink,          //IAdviseSink pointer, or NULL, indicating default data format caching
  DWORD FAR* rgdwConnection,         //Location to return array of dwConnection values
  LPCLIENTSITE pClientSite,          //Pointer to primary interface the object will use to request services
  LPSTORAGE pStg,                    //Pointer to storage to use for object
  LPVOID FAR* ppvObj                 ///Indirect pointer to location to return riid interface
);
```

Parameters

pSrcDataObj

Pointer to the data transfer object holding the new data used to create the new object. (see **OleCreateFromData**).

riid

Reference to the identifier of the interface of the object to return.

dwFlags

Value taken from the **OLECREATE** enumeration.

renderopt

Value taken from the **RENDEROPT** enumeration.

cFormats

When *renderopt* is OLERENDER_FORMAT, indicates the number of **FORMATETC** structures in the *rgFormatEtc* array, which must be at least one. In all other cases, this parameter must be zero.

rgAdvf

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **DWORD** elements, each of which is a combination of values from the **ADVF** enumeration. Each element of this array is passed in as the *advf* parameter to a call to either **IOleCache::Cache** or **IDataObject::DAdvise**, depending on whether *pAdviseSink* is NULL or non-NULL (see below). In all other cases, this parameter must be NULL.

rgFormatEtc

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **FORMATETC** structures. When *pAdviseSink* is NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to the object's **IOleCache::Cache**. This populates the data and presentation cache managed by

the object's in-process handler (typically the default handler) with presentation or other cacheable data. When *pAdviseSink* is non-NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to **IDataObject::DAdvise**. This allows the caller (typically an OLE Container) to do its own caching or processing of data received from the object.

pAdviseSink

When *renderopt* is OLERENDER_FORMAT, may be either a valid **IAdviseSink** pointer, indicating custom caching or processing of data advises, or NULL, indicating default caching of data formats.

rgdwConnection

Location to return the array of *dwConnection* values returned when the *pAdviseSink* interface is registered for each advisory connection using **IDataObject::DAdvise**, or NULL if the returned advisory connections are not needed. Must be NULL, if *pAdviseSink* is NULL.

pClientSite

Pointer to the primary interface through which the object will request services from its container. This parameter may be NULL, in which case it is the caller's responsibility to establish the client site as soon as possible using **IOleObject::SetClientSite**.

pStg

Pointer to the storage to use for the object and any default data or presentation caching established for it.

ppvObj

Location to return the *riid* interface of the newly created object.

Return Values

S_OK

Success.

E_NOINTERFACE

The object does not support the *riid* interface.

E_INVALIDARG

One or more arguments are invalid.

Remarks

The following call to **OleCreateFromData**:

```
    OleCreateFromData(lpszFileName, riid, renderopt, pFormatEtc,
pClientSite, pStg, ppvObj);
```

is equivalent to the following call to **OleCreateFromDataEx**:

```
    DWORD    advf = ADVF_PRIMEFIRST;
    OleCreateFromFileEx(rclsid, lpszFileName, riid, renderopt, 1, &advf,
pFormatEtc, NULL, pClientSite, pStg, ppvObj);
```

Existing instantiation functions (**OleCreate**, **OleCreateFromFile**, **OleCreateFromData**, **OleCreateLink**, **OleCreateLinkToFile**, and **OleCreateLinkFromData**) create only a single presentation or data format cache in the default cache location (within the '\001OlePresXXX' streams of the passed-in **IStorage**) during instantiation. Plus, these caches must be created when the object next enters the running state. Since most applications require caching at least two presentations (screen and printer) and may require caching data in a different format or location from the handler, applications must typically launch and shut

down the object server multiple times in order to prime their data caches during object creation, i.e., Insert Object, Insert Object from File, and Paste Object.

Extended versions of these creation functions solve this problem. **OleCreateEx**, **OleCreateFromFileEx**, **OleCreateFromDataEx**, **OleCreateLinkEx**, **OleCreateLinkToFileEx**, and **OleCreateLinkFromDataEx**, contain the following new parameters: *dwFlags* to indicate additional options, *cFormats* to indicate how many formats to cache, *rgAdvf*, from the **ADVF** enumeration, to specify the advise flags for each format to be cached, *pAdviseSink* to indicate whether presentation (default-handler) or data (non-default-handler) caching is required, *rgdwConnection* to return **IDataObject::DAdvise** cookies, and *pFormatEtc*, an array of formats rather than a single format.

Containers requiring that multiple presentations be cached on their behalf by the object's handler can simply call these functions and specify the number of formats in *cFormats*, the **ADVF** flags for each format in *rgAdvf*, and the set of formats in *pFormatEtc*. These containers pass NULL for *pAdviseSink*.

Containers performing all their own data- or presentation-caching perform these same steps, but pass a non-NULL *pAdviseSink*. They perform their own caching or manipulation of the object or data during **IAdviseSink::OnDataChange**. Typically, such containers never establish the advisory connections with ADVF_NODATA, although they are not prevented from doing so.

These new functions are for OLE Compound Documents. Using these functions, applications can avoid the repeated launching and initialization steps required by the current functions. They are targeted at OLE Compound Document container applications that use default data- and presentation-caching, and also at applications that provide their own caching and data transfer from the underlying **IDataObject::DAdvise** support.

See Also

OleCreateFromData, **IOleCache::Cache**, **IDataObject::DAdvise**, **IStorage**, **IAdviseSink::OnDataChange**, **IOleObject::SetClientSite**, **OLECREATE**, **OLERENDER**, **FORMATETC**, **ADVF**

OleCreateFromFile Quick Info

Creates an embedded object from the contents of a named file.

```
WINOLEAPI OleCreateFromFile(
  REFCLSID rclsid,                  //Reserved. Must be CLSID_NULL
  LPCOLESTR lpszFileName,           //Pointer to full path of file used to create object
  REFIID riid,                      //Reference to the identifier of the interface to be used to communicate with new ob
  DWORD renderopt,                  //Value from OLERENDER
  LPFORMATETC pFormatEtc,           //Pointer to the FORMATETC structure
  LPOLECLIENTSITE pClientSite,      //Pointer to an interface
  LPSTORAGE pStg,                   //Pointer tothe interface to be used as object storage
  LPVOID FAR* ppvObj                //Indirect pointer to the interface requested in riid
);
```

Parameters

rclsid

[in] Reserved. Must be CLSID_NULL.

lpszFileName

[in] Pointer to a string specifying the full path of the file from which the object should be initialized.

riid

[in] Reference to the identifier of the interface the caller later uses to communicate with the new object (usually **IID_IOleObject**, defined in the OLE headers as the interface ID of **IOleObject**).

renderopt

[in] Value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. The **OLERENDER** value chosen affects the possible values for the *pFormatEtc* parameter.

pFormatEtc

[in] Depending on which of the **OLERENDER** flags is used as the value of *renderopt*, pointer to one of the **FORMATETC** enumeration values. Refer also to the **OLERENDER** enumeration for restrictions.

pClientSite

[in] Pointer to an instance of **IOleClientSite**, the primary interface through which the object will request services from its container. May be NULL.

pStg

[in] Pointer to the **IStorage** interface on the storage object. This parameter may not be NULL.

ppvObj

[out] Indirect pointer to the interface requested in *riid* on the newly created object on return.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

Embedded object successfully created.

STG_E_FILENOTFOUND

File not bound.

OLE_E_CANT_BINDTOSOURCE

Not able to bind to source.

STG_E_MEDIUMFULL

The medium is full.

DV_E_TYMED

Invalid TYMED.

DV_E_LINDEX

Invalid LINDEX.

DV_E_FORMATETC

Invalid **FORMATETC** structure.

Remarks

The **OleCreateFromFile** function creates a new embedded object from the contents of a named file. If the ProgID in the registration database contains the PackageOnFileDrop key, it creates a package. If not, the function calls the **GetClassFile** function to get the CLSID associated with the *lpszFileName* parameter, and then creates an OLE 2-embedded object associated with that CLSID. The *rclsid* parameter of **OleCreateFromFile** will always be ignored, and should be set to CLSID_NULL.

As for other **OleCreate***Xxx* functions, the newly created object is not shown to the user for editing, which requires a **DoVerb** operation. It is used to implement insert file operations, such as the Create from File command in Word for Windows.

See Also

GetClassFile

OleCreateFromFileEx Quick Info

Extends **OLeCreateFromFile** functionality by supporting more efficient instantiation of objects in containers requiring caching of multiple presentation formats or data, instead of the single format supported by **OleCreateFromFile**.

```
HRESULT OleCreateFromFileEx(
  REFCLSID rclsid,                  //Reserved; must be CLSID_NULL
  LPCOLESTR lpszFileName,           //Pointer to name of file to initialize new object from
  REFIID riid,                      //Reference to the identifier of the interface of object to return
  DWORD dwFlags,                    //Value from OLECREATE enumeration
  DWORD renderopt,                  //Value from OLERENDER enumeration
  ULONG cFormats,                   //Number of FORMATETC structures in the rgFormatEtc array
  DWORD rgAdvf,                     //Points to an array of cFormats DWORD elements
  LPFORMATETC rgFormatEtc,          //Points to an array of cFormats FORMATETC structures
  LPADVISESINK pAdviseSink,         //IAdviseSink pointer (custom caching), or NULL (default caching)
  DWORD FAR* rgdwConnection,        //Location to return the array of dwConnection values
  LPCLIENTSITE pClientSite,         //Pointer to primary interface the object will use to request services
  LPSTORAGE pStg,                   //Pointer to storage to use for object
  LPVOID FAR* ppvObj                //Indirect pointer to location to return riid interface
);
```

Parameters

rclsid

Reserved for future use; must be CLSID_NULL.

lpszFileName

Pointer to the name of the file from which the new object should be initialized.

riid

Reference to the identifier of the interface of the object to return.

dwFlags

Value taken from the **OLECREATE** enumeration.

renderopt

Value taken from the **OLERENDER** enumeration.

cFormats

When *renderopt* is OLERENDER_FORMAT, indicates the number of FORMATETC structures in the *rgFormatEtc* array, which must be at least one. In all other cases, this parameter must be zero.

rgAdvf

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* DWORD elements, each of which is a combination of values from the **ADVF** enumeration. Each element of this array is passed in as the *advf* parameter to a call to either **IOleCache::Cache** or **IDataObject::DAdvise**, depending on whether *pAdviseSink* is NULL or non-NULL (see below). In all other cases, this parameter must be NULL.

rgFormatEtc

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* FORMATETC structures. When *pAdviseSink* is NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to the object's **IOleCache::Cache**. This populates the data and presentation cache managed by the objects in-process handler (typically the default handler) with presentation or other cacheable data. When *pAdviseSink* is non-NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to **IDataObject::DAdvise**. This allows the caller (typically an OLE Container) to do its own caching or processing of data received from the object.

pAdviseSink

When *renderopt* is OLERENDER_FORMAT, may be either a valid **IAdviseSink** pointer, indicating custom caching or processing of data advises, or NULL, indicating default caching of data formats.

rgdwConnection

Location to return the array of *dwConnection* values returned when the *pAdviseSink* interface is registered for each advisory connection using **IDataObject::DAdvise**, or NULL if the returned advisory connections are not needed. Must be NULL, if *pAdviseSink* is NULL.

pClientSite

Pointer to the primary interface through which the object will request services from its container. This parameter may be NULL, in which case it is the caller's responsibility to establish the client site as soon as possible using **IOleObject::SetClientSite**.

pStg

Pointer to the storage to use for the object and any default data or presentation caching established for it.

ppvObj

Location to return the *riid* interface of the newly created object.

Return Values

S_OK

Success.

E_NOINTERFACE

The object does not support the *riid* interface.

E_INVALIDARG

One or more arguments are invalid.

Remarks

The following call to **OleCreateFromFile**:

```
    OleCreateFromFile(rclsid, lpszFileName, riid, renderopt, pFormatEtc,
pClientSite, pStg, ppvObj);
```

is equivalent to the following call to **OleCreateFromFileEx**:

```
    DWORD    advf = ADVF_PRIMEFIRST;
    OleCreateFromFileEx(rclsid, lpszFileName, riid, renderopt, 1, &advf,
pFormatEtc, NULL, pClientSite, pStg, ppvObj);
```

Existing instantiation functions (**OleCreate**, **OleCreateFromFile**, **OleCreateFromData**, **OleCreateLink**, **OleCreateLinkToFile**, and **OleCreateLinkFromData**) create only a single presentation or data format cache in the default cache location (within the '\001OlePresXXX' streams of the passed-in **IStorage**), during instantiation. Plus, these caches must be created when the object next enters the running state.

Since most applications require caching at least two presentations (screen and printer) and may require caching data in a different format or location from the handler, applications must typically launch and shut down the object server multiple times in order to prime their data caches during object creation, i.e., Insert Object, Insert Object from File, and Paste Object.

Extended versions of these creation functions solve this problem. **OleCreateEx**, **OleCreateFromFileEx**, **OleCreateFromDataEx**, **OleCreateLinkEx**, **OleCreateLinkToFileEx**, and **OleCreateLinkFromDataEx**, contain the following new parameters: *dwFlags* to indicate additional options, *cFormats* to indicate how many formats to cache, *rgAdvf*, from the **ADVF** enumeration, to specify the advise flags for each format to be cached, *pAdviseSink* to indicate whether presentation (default-handler) or data (non-default-handler) caching is required, *rgdwConnection* to return **IDataObject::DAdvise** cookies, and *pFormatEtc*, an array of formats rather than a single format.

Containers requiring that multiple presentations be cached on their behalf by the object's handler can simply call these functions and specify the number of formats in *cFormats*, the **ADVF** flags for each format in *rgAdvf*, and the set of formats in *pFormatEtc*. These containers pass NULL for *pAdviseSink*.

Containers performing all their own data- or presentation-caching perform these same steps, but pass a non-NULL *pAdviseSink*. They perform their own caching or manipulation of the object or data during **IAdviseSink::OnDataChange**. Typically, such containers never establish the advisory connections with ADVF_NODATA, although they are not prevented from doing so.

These new functions are for OLE Compound Documents. Using these functions, applications can avoid the repeated launching and initialization steps required by the current functions. They are targeted at OLE Compound Document container applications that use default data- and presentation-caching, and also at applications that provide their own caching and data transfer from the underlying **IDataObject::DAdvise** support.

See Also

OleCreateFromFile, **IOleCache::Cache**, **IDataObject::DAdvise**, **IAdviseSink::OnDataChange**, **IOleObject::SetClientSite**, **IStorage**, **OLERENDER**, **FORMATETC**, **ADVF**

OleCreateLink Quick Info

Creates an OLE compound-document linked object.

```
WINOLEAPI OleCreateLink(
  LPMONIKER pmkLinkSrc,          //Pointer to moniker indicating source of linked object
  REFIID riid,                   //Reference to the identifier of the interfacer to be used to communicate with the ne
                                 object
  DWORD renderopt,               //Value from OLERENDER
  LPFORMATETC pFormatEtc,        //Pointer to a FORMATETC structure
  LPOLECLIENTSITE pClientSite,   //Pointer to an interface
  LPSTORAGE pStg,                //Pointer to the object's storage
  LPVOID FAR* ppvObj             //Indirect pointer to the interface requested in riid
);
```

Parameters

pmkLinkSrc

[in] Pointer to the **IMoniker** interface on the moniker that can be used to locate the the source of the linked object.

riid

[in] Reference to the identifier of the interface the caller later uses to communicate with the new object (usually **IID_IOleObject**, defined in the OLE headers as the interface identifier for **IOleObject**).

renderopt

[in] Specifies a value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. Additional considerations are described in the Remarks section below.

pFormatEtc

[in] Pointer to a value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. The **OLERENDER** value chosen affects the possible values for the *pFormatEtc* parameter.

pClientSite

[in] Pointer to an instance of **IOleClientSite**, the primary interface through which the object will request services from its container. May be NULL.

pStg

[in] Pointer to the **IStorage** interface on the storage object. This parameter may not be NULL.

ppvObj

[out] Indirect pointer to the interface requested in *riid* on the newly created object.

Return Values

S_OK

The compound-document linked object was created successfully.

OLE_E_CANT_BINDTOSOURCE

Not able to bind to source. Binding is necessary to get the cache's initialization data.

Remarks

Call **OleCreateLink** to allow a container to create a link to an object.

See Also

IOleObject::SetMoniker, **IOleClientSite::GetMoniker**

OleCreateLinkEx Quick Info

Extends **OleCreateLink** functionality by supporting more efficient instantiation of objects in containers requiring caching of multiple formats of presentations or data, instead of the single format supported by **OleCreateLink.**

```
HRESULT OleCreateFromLinkEx(
    LPMONIKER pmkLinkSrc,              //Pointer to a moniker to object to create link to
    REFIID riid,                       //Reference to the identifier of the interface of the link object to return
    DWORD dwFlags,                     //Value from OLECREATE enumeration
    DWORD renderopt,                   //Value from OLERENDER enumeration
    ULONG cFormats,                    //Number of FORMATETCs in rgFormatEt
    DWORD rgAdvf,                      //Points to an array of cFormats DWORD elements
    LPFORMATETC rgFormatEtc,           //Points to an array of cFormats FORMATETC structures
    LPADVISESINK pAdviseSink,          //IAdviseSink pointer, or NULL, indicating default data format caching
    DWORD FAR* rgdwConnection,         //Location to return array of dwConnection values
    LPCLIENTSITE pClientSite,          //Pointer to the primary interface the object will to use to request services
    LPSTORAGE pStg,                    //Pointer to storage to use for the object
    LPVOID FAR* ppvObj                 //Indirect pointer to location to return riid interface
);
```

Parameters

pmkLinkSrc

Pointer to a moniker to the object to create a link to.

riid

Reference to the identifier of the interface of the object to return.

dwFlags

Value taken from the **OLECREATE** enumeration.

renderopt

Value taken from the **OLERENDER** enumeration.

cFormats

When *renderopt* is OLERENDER_FORMAT, indicates the number of **FORMATETC** structures in the *rgFormatEtc* array, which must be at least one. In all other cases, this parameter must be zero.

rgAdvf

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **DWORD** elements, each of which is a combination of values from the **ADVF** enumeration. Each element of this array is passed in as the *advf* parameter to a call to either **IOleCache::Cache** or **IDataObject::DAdvise**, depending on whether *pAdviseSink* is NULL or non-NULL (see below). In all other cases, this parameter must be NULL.

rgFormatEtc

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **FORMATETC** structures. When *pAdviseSink* is NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to the object's **IOleCache::Cache**. This populates the data and presentation cache managed by the objects in-process handler (typically the default handler) with presentation or other cacheable

data. When *pAdviseSink* is non-NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to **IDataObject::DAdvise**. This allows the caller (typically an OLE Container) to do its own caching or processing of data received from the object.

pAdviseSink

When *renderopt* is OLERENDER_FORMAT, may be either a valid **IAdviseSink** pointer, indicating custom caching or processing of data advises, or NULL, indicating default caching of data formats.

rgdwConnection

Location to return the array of *dwConnection* values returned when the *pAdviseSink* interface is registered for each advisory connection using **IDataObject::DAdvise**, or NULL if the returned advisory connections are not needed. Must be NULL, if *pAdviseSink* is NULL.

pClientSite

Pointer to the primary interface through which the object will request services from its container. This parameter may be NULL, in which case it is the caller's responsibility to establish the client site as soon as possible using **IOleObject::SetClientSite**.

pStg

Pointer to the storage to use for the object and any default data or presentation caching established for it.

ppvObj

Indirect pointer to location to return the *riid* interface of the newly created object.

Return Values

S_OK

Success.

E_NOINTERFACE

The object does not support the *riid* interface.

E_INVALIDARG

One or more arguments are invalid.

Remarks

The following call to **OleCreateLink**:

```
    OleCreateLink(pmkLinkSrc, riid, renderopt, pFormatEtc, pClientSite,
pStg, ppvObj);
```

is equivalent to the following call to **OleCreateLinkEx**:

```
    DWORD    advf = ADVF_PRIMEFIRST;
    OleCreateFromFileEx(pmkLinkSrc, riid, renderopt, 1, &advf, pFormatEtc,
NULL, NULL, pClientSite, pStg, ppvObj);
```

Existing instantiation functions (**OleCreate**, **OleCreateFromFile**, **OleCreateFromData**, **OleCreateLink**, **OleCreateLinkToFile**, and **OleCreateLinkFromData**) create only a single presentation or data format cache in the default cache location (within the '\001OlePresXXX' streams of the passed-in **IStorage**) during instantiation. Plus, these caches must be created when the object next enters the running state. Since most applications require caching in at least two presentations (screen and printer) and may require caching data in a different format or location from the handler, applications must typically launch and shut down the object server multiple times in order to prime their data caches during object creation, i.e., Insert Object, Insert Object from File, and Paste Object.

Extended versions of these creation functions solve this problem. **OleCreateEx**, **OleCreateFromFileEx**, **OleCreateFromDataEx**, **OleCreateLinkEx**, **OleCreateLinkToFileEx**, and **OleCreateLinkFromDataEx**, contain the following new parameters: *dwFlags* to indicate additional options, *cFormats* to indicate how many formats to cache, *rgAdvf*, from the **ADVF** enumeration, to specify the advise flags for each format to be cached, *pAdviseSink* to indicate whether presentation (default-handler) or data (non-default-handler) caching is required, *rgdwConnection* to return **IDataObject::DAdvise** cookies, and *pFormatEtc*, an array of formats rather than a single format.

Containers requiring that multiple presentations be cached on their behalf by the object's handler can simply call these functions and specify the number of formats in *cFormats*, the **ADVF** flags for each format in *rgAdvf*, and the set of formats in *pFormatEtc*. These containers pass NULL for *pAdviseSink*.

Containers performing all their own data- or presentation-caching perform these same steps, but pass a non-NULL *pAdviseSink*. They perform their own caching or manipulation of the object or data during **IAdviseSink::OnDataChange**. Typically, such containers never establish the advisory connections with ADVF_NODATA, although they are not prevented from doing so.

These new functions are for OLE Compound Documents. Using these functions, applications can avoid the repeated launching and initialization steps required by the current functions. They are targeted at OLE Compound Document container applications that use default data- and presentation-caching, and also at applications that provide their own caching and data transfer from the underlying **IDataObject::DAdvise** support.

See Also

OleCreateLink, **IOleCache::Cache**, **IDataObject::DAdvise**, **IAdviseSink::OnDataChange**, **IOleObject::SetClientSite**, **OLECREATE**, **OLERENDER**, **FORMATETC**, **ADVF**

OleCreateLinkFromData Quick Info

Creates a linked object from a data transfer object retrieved either from the clipboard or as part of an OLE drag-and-drop operation.

```
WINOLEAPI OleCreateLinkFromData(
  LPDATAOBJECT pSrcDataObj,          //Pointer to data transfer objectt
  REFIID riid,                       //Reference to the identifier of the interface to be used to communicate with the nev
  DWORD renderopt,                   //OLERENDER value
  LPFORMATETC pFormatEtc,            //Pointer to a FORMATETC structure
  LPOLECLIENTSITE pClientSite,       //Pointer to an interface
  LPSTORAGE pStg,                    //Pointer to the object storage
  LPVOID FAR* ppvObj                 //Indirect pointer to requested object
);
```

Parameters

pSrcDataObj

[in] Pointer to the **IDataObject** interface on the data transfer object from which the linked object is to be created.

riid

[in] Reference to the identifier of interface the caller later uses to communicate with the new object (usually **IID_IOleObject**, defined in the OLE headers as the interface identifier for **IOleObject**).

renderopt

[in] Value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. Additional considerations are described in the following Remarks section.

pFormatEtc

[in] Pointer to a value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. The **OLERENDER** value chosen affects the possible values for the *pFormatEtc* parameter.

pClientSite

[in] Pointer to an instance of **IOleClientSite**, the primary interface through which the object will request services from its container. May be NULL.

pStg

[in] Pointer to the **IStorage** interface on the storage object. This parameter may not be NULL.

ppvObj

[out] Indirect pointer to the interface requested in *riid* on the newly created object on return.

Return Values

S_OK

The linked object was created successfully.

CLIPBRD_E_CANT_OPEN

Not able to open the clipboard.

OLE_E_CANT_GETMONIKER

Not able to extract the object's moniker.

OLE_E_CANT_BINDTOSOURCE

Not able to bind to source. Binding is necessary to get the cache's initialization data.

Remarks

The **OleCreateLinkFromData** function is used to implement either a paste-link or a drag-link operation. Its operation is similar to that of the **OleCreateFromData** function, except that it creates a link, and looks for different data formats. If the CF_LINKSOURCE format is not present, and either the FileName or FileNameW clipboard format is present in the data transfer object, **OleCreateLinkFromData** creates a package containing the link to the indicated file.

You use the *renderopt* and *pFormatetc* parameters to control the caching capability of the newly created object. For general information on how to determine what is to be cached, refer to the **OLERENDER** enumeration for a description of the interaction between *renderopt* and *pFormatetc*. There are, however, some additional specific effects of these parameters on the way **OleCreateLinkFromData** initializes the cache, as follows:

Value	Description
OLERENDER_DRAW, OLERENDER_FORMAT	If the presentation information is in the other formats in the source data object, this information is used. If the information is not present, the cache is initially empty, but will be filled the first time the object is run. No other formats are cached in the newly created object.
OLERENDER_NONE, OLERENDER_ASIS	Nothing is to be cached in the newly created object.

See Also

OleCreateLink

OleCreateLinkFromDataEx Quick Info

Extends **OleCreateLinkFromData** functionality by supporting more efficient instantiation of objects in containers requiring caching of multiple formats of presentations or data, instead of the single format supported by **OleCreateLinkFromData**.

```
HRESULT OleCreateLinkFromDataEx(
  LPDATAOBJECT pSrcDataObj,          //Pointer to the data object to create a link object from
  REFIID riid,                       //Reference to the identifier of the interface of the link object to return
  DWORD dwFlags,                     //Value from OLECREATE enumeration
  DWORD renderopt,                   //Value from OLERENDER enumeration
  ULONG cFormats,                    //Number of FORMATETCs in rgFormatEtc
  DWORD rgAdvf,                      //Points to an array of cFormats DWORD elements
  LPFORMATETC rgFormatEtc,           //Points to an array of cFormats FORMATETC structures
  LPADVISESINK pAdviseSink,          //IAdviseSink pointer (custom caching); NULL (default caching); NULL otherwise
  DWORD FAR* rgdwConnection,         //Location to return array of dwConnection values
  LPCLIENTSITE pClientSite,          //Pointer to the primary interface the object will use to request services
  LPSTORAGE pStg,                    //Pointer to storage to use for object
  LPVOID FAR* ppvObj                 //Indirect pointer to location to return riid interface
);
```

Parameters

pSrcDataObj

Pointer to the data object to create a link object from.

riid

Reference to the identifier of the interface of the object to return.

dwFlags

Value taken from the **OLECREATE** enumeration.

renderopt

Value taken from the **RENDEROPT** enumeration.

cFormats

When *renderopt* is OLERENDER_FORMAT, indicates the number of **FORMATETC** structures in the *rgFormatEtc* array, which must be at least one. In all other cases, this parameter must be zero.

rgAdvf

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **DWORD** elements, each of which is a combination of values from the **ADVF** enumeration. Each element of this array is passed in as the *advf* parameter to a call to either **IOleCache::Cache** or **IDataObject::DAdvise**, depending on whether *pAdviseSink* is NULL or non-NULL (see below). In all other cases, this parameter must be NULL.

rgFormatEtc

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **FORMATETC** structures. When *pAdviseSink* is NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to the object's **IOleCache::Cache**. This populates the data and presentation cache managed by the objects in-process handler (typically the default handler) with presentation or other cacheable

data. When *pAdviseSink* is non-NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to **IDataObject::DAdvise**. This allows the caller (typically an OLE Container) to do its own caching or processing of data received from the object.

pAdviseSink

When *renderopt* is OLERENDER_FORMAT, may be either a valid **IAdviseSink** pointer, indicating custom caching or processing of data advises, or NULL, indicating default caching of data formats.

rgdwConnection

Location to return the array of *dwConnection* values returned when the *pAdviseSink* interface is registered for each advisory connection using **IDataObject::DAdvise**, or NULL if the returned advisory connections are not needed. Must be NULL, if *pAdviseSink* is NULL.

pClientSite

Pointer to the primary interface through which the object will request services from its container. This parameter may be NULL, in which case it is the caller's responsibility to establish the client site as soon as possible using **IOleObject::SetClientSite**.

pStg

Pointer to the storage to use for the object and any default data or presentation caching established for it.

ppvObj

Indirect pointer to location to return the *riid* interface of the newly created object.

Return Values

S_OK

Success.

E_NOINTERFACE

The object does not support the *riid* interface.

E_INVALIDARG

One or more arguments are invalid.

Remarks

The following call to **OleCreateLinkFromData**:

```
    OleCreateLinkFromData(pSrcDataObj, riid, renderopt, pFormatEtc,
pClientSite, pStg, ppvObj);
```

is equivalent to the following call to **OleCreateLinkFromDataEx**:

```
    DWORD    advf = ADVF_PRIMEFIRST;
    OleCreateLinkFromDataEx(pSrcDataObj, riid, renderopt, 1, &advf,
pFormatEtc, NULL, NULL, pClientSite, pStg, ppvObj);
```

Existing instantiation functions (**OleCreate**, **OleCreateFromFile**, **OleCreateFromData**, **OleCreateLink**, **OleCreateLinkToFile**, and **OleCreateLinkFromData**), create only a single presentation or data format cache in the default cache location (within the '\001OlePresXXX' streams of the passed-in **IStorage**) during instantiation. Plus, these caches must be created when the object next enters the running state. Since most applications require caching at least two presentations (screen and printer) and may require caching data in a different format or location from the handler, applications must typically launch and shut down the object server multiple times in order to prime their data caches during object creation, i.e., Insert

Object, Insert Object from File, and Paste Object.

Extended versions of these creation functions solve this problem. **OleCreateEx**, **OleCreateFromFileEx**, **OleCreateFromDataEx**, **OleCreateLinkEx**, **OleCreateLinkToFileEx**, and **OleCreateLinkFromDataEx**, contain the following new parameters: *dwFlags* to indicate additional options, *cFormats* to indicate how many formats to cache, *rgAdvf*, from the **ADVF** enumeration, to specify the advise flags for each format to be cached, *pAdviseSink* to indicate whether presentation (default-handler) or data (non-default-handler) caching is required, *rgdwConnection* to return **IDataObject::DAdvise** cookies, and *pFormatEtc*, an array of formats rather than a single format.

Containers requiring that multiple presentations be cached on their behalf by the object's handler can simply call these functions and specify the number of formats in *cFormats*, the **ADVF** flags for each format in *rgAdvf*, and the set of formats in *pFormatEtc*. These containers pass NULL for *pAdviseSink*.

Containers performing all their own data- or presentation-caching perform these same steps, but pass a non-NULL *pAdviseSink*. They perform their own caching or manipulation of the object or data during **IAdviseSink::OnDataChange**. Typically such containers never establish the advisory connections with ADVF_NODATA, although they are not prevented from doing so.

These new functions are for OLE Compound Documents. Using these functions, applications can avoid the repeated launching and initialization steps required by the current functions. They are targeted at OLE Compound Document container applications that use default data- and presentation-caching, and also at applications that provide their own caching and data transfer from the underlying **IDataObject::DAdvise** support.

See Also

OleCreateLinkFromData, **IOleCache::Cache**, **IDataObject::DAdvise**, **IAdviseSink::OnDataChange**, **IOleObject::SetClientSite**, **OLECREATE**, **OLERENDER**, **FORMATETC**, **ADVF**

OleCreateLinkToFile Quick Info

Creates an object that is linked to a file.

```
WINOLEAPI OleCreateLinkToFile(
  LPWSTR lpszFileName,          //Pointer to source of linked object
  REFIID riid,                  //Reference to the identifier of the interface to be used to communicate with the new o
  DWORD renderopt,              //Value from OLERENDER
  LPFORMATETC pFormatEtc,       //Pointer to a FORMATETC structure
  IOleClientSite * pClientSite, //Pointer to an interface
  IStorage * pStg,              //Pointer to the object's storage
  void ** ppvObj                //Indirect pointer to the interface requested in riid
);
```

Parameters

lpszFileName

[in] Pointer to a string naming the source file to be linked to.

riid

[in] Reference to the identifier of the interface the caller later uses to communicate with the new object (usually **IID_IOleObject**, defined in the OLE headers as the interface identifier for **IOleObject**).

renderopt

[in] Value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. Additional considerations are described in the following Remarks section.

pFormatEtc

[in] Pointer to a value from the enumeration **OLERENDER** that indicates the locally cached drawing or data-retrieval capabilities the newly created object is to have. The **OLERENDER** value chosen affects the possible values for the *pFormatEtc* parameter.

pClientSite

[in] Pointer to an instance of **IOleClientSite**, the primary interface through which the object will request services from its container. May be NULL.

pStg

[in] Pointer to the **IStorage** interface on the storage object. This parameter may not be NULL.

ppvObj

[out] Indirect pointer to the interface requested in *riid* on the newly created object on return.

Return Values

S_OK

The object was created successfully.

STG_E_FILENOTFOUND

The file name is invalid.

OLE_E_CANT_BINDTOSOURCE

Not able to bind to source.

Remarks

The **OleCreateLinkToFile** function differs from the **OleCreateLink** function because it can create links both to files that are not aware of OLE, as well as to those that are using the Windows Packager.

See Also

OleCreateLink

OleCreateLinkToFileEX

Extends **OleCreateLinkToFile** functionality by supporting more efficient instantiation of objects in containers requiring caching of multiple formats of presentations or data, instead of the single format supported by **OleCreateLinkToFile**.

```
HRESULT OleCreateLinkToFileEx(
  LPCOLESTR lpszFileName,          //Pointer to the name of the file to create a link to.
  REFIID riid,                     //Reference to the identifier of the interface of the link object to return
  DWORD dwFlags,                   //Value from OLECREATE enumeration
  DWORD renderopt,                 //Value from OLERENDER enumeration
  ULONG cFormats,                  //Number of FORMATETCs in rgFormatEtc
  DWORD rgAdvf,                    //Points to an array of cFormats DWORD elements
  LPFORMATETC rgFormatEtc,         //Points to an array of cFormats FORMATETC structures
  LPADVISESINK pAdviseSink,        ///IAdviseSink pointer (custom caching); NULL (default caching); NULL otherwise
  DWORD FAR* rgdwConnection,       //Location to return array of dwConnection values
  LPCLIENTSITE pClientSite,        //Pointer to the primary interface the object will use to request services.
  LPSTORAGE pStg,                  //Pointer to storage to use for object
  LPVOID FAR* ppvObj               //Indirect pointer to location to return riid interface
);
```

Parameters

lpszFileName

Pointer to the name of the file to create a link to.

riid

Reference to the identifier of the interface of the object to return.

dwFlags

Value taken from the **OLECREATE** enumeration.

renderopt

Value taken from the **RENDEROPT** enumeration.

cFormats

When *renderopt* is OLERENDER_FORMAT, indicates the number of **FORMATETC** structures in the *rgFormatEtc* array, which must be at least one. In all other cases, this parameter must be zero.

rgAdvf

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **DWORD** elements, each of which is a combination of values from the **ADVF** enumeration. Each element of this array is passed in as the *advf* parameter to a call to either **IOleCache::Cache** or **IDataObject::DAdvise**, depending on whether *pAdviseSink* is NULL or non-NULL (see below). In all other cases, this parameter must be NULL.

rgFormatEtc

When *renderopt* is OLERENDER_FORMAT, points to an array of *cFormats* **FORMATETC** structures. When *pAdviseSink* is NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to the object's **IOleCache::Cache**. This populates the data and presentation cache managed by the objects in-process handler (typically the default handler) with presentation or other cacheable

data. When *pAdviseSink* is non-NULL, each element of this array is passed as the *pFormatEtc* parameter to a call to **IDataObject::DAdvise**. This allows the caller (typically an OLE Container) to do its own caching or processing of data received from the object.

pAdviseSink

When *renderopt* is OLERENDER_FORMAT, may be either a valid **IAdviseSink** pointer, indicating custom caching or processing of data advises, or NULL, indicating default caching of data formats.

rgdwConnection

Location to return the array of *dwConnection* values returned when the *pAdviseSink* interface is registered for each advisory connection using **IDataObject::DAdvise**, or NULL if the returned advisory connections are not needed. Must be NULL, if *pAdviseSink* is NULL.

pClientSite

Pointer to the primary interface through which the object will request services from its container. This parameter may be NULL, in which case it is the caller's responsibility to establish the client site as soon as possible using **IOleObject::SetClientSite**.

pStg

Pointer to the storage to use for the object and any default data or presentation caching established for it.

ppvObj

Location to return the *riid* interface of the newly created object.

Return Values

S_OK

Success.

E_NOINTERFACE

The object does not support the *riid* interface.

E_INVALIDARG

One or more arguments are invalid.

Remarks

The following call to **OleCreateLinkToFile**:

```
    OleCreateLinkToFile(lpszFileName, riid, renderopt, pFormatEtc,
pClientSite, pStg, ppvObj);
```

is equivalent to the following call to **OleCreateLinkToFileEx**:

```
    DWORD    advf = ADVF_PRIMEFIRST;
    OleCreateLinkToFileEx(lpszFileName, riid, renderopt, 1, &advf,
pFormatEtc, NULL, NULL, pClientSite, pStg, ppvObj);
```

Existing instantiation functions (**OleCreate**, **OleCreateFromFile**, **OleCreateFromData**, **OleCreateLink**, **OleCreateLinkToFile**, and **OleCreateLinkFromData**) create only a single presentation or data format cache in the default cache location (within the '\001OlePresXXX' streams of the passed-in **IStorage**) during instantiation. Plus, these caches must be created when the object next enters the running state.

Since most applications require caching at least two presentations (screen and printer) and may require caching data in a different format or location from the handler, applications must typically launch and shut down the object server multiple times in order to prime their data caches during object creation, i.e., Insert Object, Insert Object from File, and Paste Object.

Extended versions of these creation functions solve this problem. **OleCreateEx**, **OleCreateFromFileEx**, **OleCreateFromDataEx**, **OleCreateLinkEx**, **OleCreateLinkToFileEx**, and **OleCreateLinkFromDataEx**, contain the following new parameters: *dwFlags* to indicate additional options, *cFormats* to indicate how many formats to cache, *rgAdvf*, from the **ADVF** enumeration, to specify the advise flags for each format to be cached, *pAdviseSink* to indicate whether presentation (default-handler) or data (non-default-handler) caching is required, *rgdwConnection* to return **IDataObject::DAdvise** cookies, and *pFormatEtc*, an array of formats rather than a single format.

Containers requiring that multiple presentations be cached on their behalf by the object's handler can simply call these functions and specify the number of formats in *cFormats*, the **ADVF** flags for each format in *rgAdvf*, and the set of formats in *pFormatEtc*. These containers pass NULL for *pAdviseSink*.

Containers performing all their own data- or presentation-caching perform these same steps, but pass a non-NULL *pAdviseSink*. They perform their own caching or manipulation of the object or data during **IAdviseSink::OnDataChange**. Typically, such containers never establish the advisory connections with ADVF_NODATA, although they are not prevented from doing so.

These new functions are for OLE Compound Documents. Using these functions, applications can avoid the repeated launching and initialization steps required by the current functions. They are targeted at OLE Compound Document container applications that use default data- and presentation-caching, and also at applications that provide their own caching and data transfer from the underlying **IDataObject::DAdvise** support.

See Also

OleCreateLinkToFile, **IOleCache::Cache**, **IDataObject::DAdvise**, **IStorage**, **IAdviseSink::OnDataChange**, **IOleObject::SetClientSite**, **OLECREATE**, **OLERENDER**, **FORMATETC**, **ADVF**

OleCreateMenuDescriptor Quick Info

Creates and returns an OLE menu descriptor (that is, an OLE-provided data structure that describes the menus) for OLE to use when dispatching menu messages and commands.

```
HOLEMENU OleCreateMenuDescriptor(

  HMENU hmenuCombined,                  //Handle to the combined menu
  LPOLEMENUGROUPWIDTHS lpMenuWidths     //Pointer to the number of menus in each group
);
```

Parameters

hmenuCombined

[in] Handle to the combined menu created by the object.

lpMenuWidths

[in] Pointer to an array of six LONG values giving the number of menus in each group.

Return Value

Returns the handle to the descriptor, or NULL if insufficient memory is available.

Remarks

The **OleCreateMenuDescriptor** function can be called by the object to create a descriptor for the composite menu. OLE then uses this descriptor to dispatch menu messages and commands. To free the shared menu descriptor when it is no longer needed, the container should call the companion helper function, **OleDestroyMenuDescriptor**.

See Also

OleDestroyMenuDescriptor

OleCreatePictureIndirect Quick Info

Creates a new picture object initialized according to a **PICTDESC** structure, which can be NULL to create an uninitialized object if the caller wishes to have the picture initialize itself through **IPersistStream::Load**. The *fOwn* parameter indicates whether the picture is to own the GDI picture handle for the picture it contains, so that the picture object will destroy its picture when the object itself is destroyed. The function returns an interface pointer to the new picture object specified by the caller in the *riid* parameter. A **QueryInterface** is built into this call. The caller is responsible for calling **Release** through the interface pointer returned.

```
STDAPI OleCreatePictureIndirect(
  PICTDESC* pPictDesc,      //Pointer to the structure of parameters for picture
  REFIID  riid,             //Reference to the identifier of the interface
  BOOL fOwn,                //Whether the picture is to be destroyed
  VOID** ppvObj             //Indirect pointer to the initial interface pointer on the new object
);
```

Parameters

pPictDesc

[in] Pointer to a caller-allocated structure containing the initial state of the picture.

riid

[in] Reference to the identifier of the interface describing the type of interface pointer to return in *ppvObj*.

fOwn

[in] If TRUE, the picture object is to destroy its picture when the object is destroyed. If FALSE, the caller is responsible for destroying the picture.

ppvObj

[out] Indirect pointer to the initial interface pointer on the new object. If the call is successful, the caller is responsible for calling **Release** through this interface pointer when the new object is no longer needed. If the call fails, the value of *ppvObj* is set to NULL.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The new picture object was created successfully.

E_NOINTERFACE

The object does not support the interface specified in *riid*.

E_POINTER

The address in *pPictDesc* or *ppvObj* is not valid. For example, it may be NULL.

See Also

OleLoadPicture, **PICTDESC**

OleCreatePropertyFrame Quick Info

Invokes a new property frame, that is, a property sheet dialog box, whose parent is *hwndOwner,* where the dialog is positioned at the point (*x,y*) in the parent window and has the caption *lpszCaption*.

```
STDAPI OleCreatePropertyFrame(
  HWND hwndOwner,              //Parent window of property sheet dialog box
  UINT x,                      //Horizontal position for dialog box
  UINT y,                      //Vertical position for dialog box
  LPCOLESTR lpszCaption,       //Pointer to the dialog box caption
  ULONG cObjects,              //Number of object pointers in lplpUnk
  LPUNKNOWN FAR* lplpUnk,      //Pointer to the objects for property sheet
  ULONG cPages,                //Number of property pages in lpPageClsID
  LPCLSID lpPageClsID,         //Array of CLSIDs for each property page
  LCID lcid,                   //Locale identifier for property sheet locale
  DWORD dwReserved,            //Reserved
  LPVOID lpvReserved           //Reserved
);
```

Parameters

hwndOwner

[in] Parent window of the resulting property sheet dialog box.

x

[in] Horizontal position for the dialog box relative to *hwndOwner*.

y

[in] Vertical position for the dialog box relative to *hwndOwner*.

lpszCaption

[in] Pointer to the string used for the caption of the dialog box.

cObjects

[in] Number of object pointers passed in *lplpUnk*.

lplpUnk

[in] An array of **IUnknown** pointers on the objects for which this property sheet is being invoked. The number of elements in the array is specified by *cObjects*. These pointers are passed to each property page through **IPropertyPage::SetObjects**.

cPages

[in] Number of property pages specified in *lpPageClsID*.

lpPageClsID

[in] Array of size *cPages* containing the **CLSID**s of each property page to display in the property sheet.

lcid

[in] Locale identifier to use for the property sheet. Property pages can retrieve this identifier through

IPropertyPageSite::GetLocaleID.

dwReserved

[in] Reserved for future use; must be zero.

lpvReserved

[in] Reserved for future use; must be NULL.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The dialog box was invoked and operated successfully.

E_POINTER

The address in *lpszCaption*, *lplpUnk*, or *lpPageClsID* is not valid. For example, any one of them may be NULL.

Remarks

The property pages to be displayed are identified with *lpPageClsID,* which is an array of *cPages* **CLSID** values. The objects that are affected by this property sheet are identified in *lplpUnk*, an array of size *cObjects* containing **IUnknown** pointers.

This function always creates a modal dialogbox and does not return until the dialog box is closed.

See Also

OleCreatePropertyFrameIndirect

OleCreatePropertyFrameIndirect Quick Info

Creates a property frame, that is, a property sheet dialog box, based on a structure (**OCPFIPARAMS**) that contains the parameters, rather than specifying separate parameters as when calling **OleCreatePropertyFrame.**

```
STDAPI OleCreatePropertyFrameIndirect(

  OCPFIPARAMS* pParams      //Pointer to the structure of parameters for dialog box
);
```

Parameters

pParams

[in] Pointer to the caller-allocated structure containing the creation parameters for the dialog box.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The dialog box was invoked and operated successfully.

E_POINTER

The address in *pParams* is not valid. For example, it may be NULL.

Remarks

Besides *cbStructSize* (the size of the structure) and *dispIDInitialProperty*, all of the members of the **OCPFIPARAMS** structure have the same semantics as the parameters for **OleCreatePropertyFrame**. When *dispIDInitialProperty* is DISPID_UNKNOWN, the behavior of the two functions is identical.

Working in conjunction with **IPerPropertyBrowsing** and **IPropertyPage2**, *dispIDInitialProperty* allows the caller to specify which single property should be highlighted when the dialog box is made visible. This feature is not available when using **OleCreatePropertyFrame**. To determine the page and property to show initially, the property frame will do the following:

1. Call (*lplpUnk)->QueryInterface(IID_IPerPropertyBrowsing, ...) to get an interface pointer to the first object.
2. Call IPerPropertyBrowsing::MapPropertyToPage(dispIDInitialProperty, ...) to determine which page **CLSID** contains the property to be highlighted. All objects for which this frame is being invoked must support the set of properties displayed in the frame.
3. When the dialog box is created, the property page with the **CLSID** retrieved in Step 2 is activated with **IPropertyPage::Activate**.
4. The property frame queries the active page for **IPropertyPage2**.
5. If successful, the frame calls IPropertyPage2::EditProperty(dispIDInitialProperty) to highlight the correct field in that dialog box.

See Also

OCPFIPARAMS, **OleCreatePropertyFrame**

OleCreateStaticFromData Quick Info

Creates a static object (containing only a representation, with no native data) from a data transfer object.

```
WINOLEAPI OleCreateStaticFromData(
  LPDATAOBJECT pSrcDataObj,          //Pointer to the data transfer object
  REFIID riid,                       //Reference to the identifier of the interface to be used to communicate with the nev
  DWORD renderopt,                   //Value from OLERENDER
  LPFORMATETC pFormatEtc,            //Depending on renderopt, pointer to value from FORMATETC
  LPOLECLIENTSITE pClientSite,       //Pointer to the interface
  LPSTORAGE pStg,                    //Pointer to store object
  LPVOID FAR* ppvObj                 //Indirect pointer to the interface requested in riid
);
```

Parameters

pSrcDataObj

[in] Pointer to the **IDataObject** interface on the data transfer object that holds the data from which the object will be created.

riid

[in] Reference to the identifier of the interface with which the caller is to communicate with the new object (usually **IID_IOleObject**, defined in the OLE headers as the interface identifier for **IOleObject**).

renderopt

[in] Value from the enumeration **OLERENDER** indicating the locally cached drawing or data-retrieval capabilities that the container wants in the newly created component. It is an error to pass the render options OLERENDER_NONE or OLERENDER_ASIS to this function.

pFormatEtc

[in] Depending on which of the **OLERENDER** flags is used as the value of *renderopt*, may be a pointer to one of the **FORMATETC** enumeration values. Refer to the **OLERENDER** enumeration for restrictions.

pClientSite

[in] Pointer to an instance of **IOleClientSite**, the primary interface through which the object will request services from its container. May be NULL.

pStg

[in] Pointer to the **IStorage** interface for storage for the object. This parameter may not be NULL.

ppvObj

[out] When the function returns successfully, indirect pointer to the interface requested in *riid* on the newly created object.

Return Value

S_OK

The object was successfully created.

Remarks

The **OleCreateStaticFromData** function can convert any object, as long as it provides an **IDataObject** interface, to a static object. It is useful in implementing the Convert To Picture option for OLE linking or embedding.

Static objects can be created only if the source supports one of the OLE-rendered clipboard formats: CF_METAFILEPICT, CF_DIB, or CF_ BITMAP, and CF_ENHMF.

You can also call **OleCreateStaticFromData** to paste a static object from the clipboard. To determine whether an object is static, call the **OleQueryCreateFromData** function, which returns OLE_S_STATIC if one of CF_METAFILEPICT, CF_DIB, or CF_BITMAP is present and an OLE format is not present. This indicates that you should call **OleCreateStaticFromData** rather than the **OleCreateFromData** function to create the object.

The new static object is of class CLSID_StaticMetafile (in the case of CF_METAFILEPICT) and CLSID_StaticDib (in the case of CF_DIB or CF_BITMAP). The static object sets the OLEMISC_STATIC and OLE_CANTLINKINSIDE bits returned from **IOleObject::GetMiscStatus**. The static object will have the aspect DVASPECT_CONTENT and a LINDEX of -1.

The *pDataObject* is still valid after **OleCreateStaticFromData** returns. It is the caller's responsibility to free *pDataObject* – OLE does not release it.

There cannot be more than one presentation stream in a static object.

Note The **OLESTREAM**<->**IStorage** conversion functions also convert static objects.

See Also

OleCreateFromData

OleDestroyMenuDescriptor Quick Info

Called by the container to free the shared menu descriptor allocated by the **OleCreateMenuDescriptor** function.

```
void OleDestroyMenuDescriptor(

  HOLEMENU holemenu      //Handle to the shared menu descriptor
);
```

Parameter

holemenu

[in] Handle to the shared menu descriptor that was returned by the **OleCreateMenuDescriptor** function.

Return Value

None. (This function does not indicate failure.)

See Also

OleCreateMenuDescriptor

OleDoAutoConvert Quick Info

Automatically converts an object to a new class if automatic conversion for that object class is set in the registry.

```
WINOLEAPI OleDoAutoConvert(

  IStorage * pStg,          //Pointer to storage object to be converted
  LPCLSID pClsidNew         //Pointer to new CLSID of converted object
);
```

Parameters

pStg

[in] Pointer to the **IStorage** interface on the storage object to be converted.

pClsidNew

[out] Pointserto the new CLSID for the object being converted. If there was no automatic conversion, this may be the same as the original class.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

No conversion is needed or a conversion was successfully completed.

REGDB_E_KEYMISSING

The function cannot read a key from the registry.

This function can also return any of the error values returned by the **OleGetAutoConvert** function. When accessing storage and stream objects, see the **IStorage::OpenStorage** and **IStorage::OpenStream** methods for possible errors. When it is not possible to determine the existing CLSID or when it is not possible to update the storage object with new information, see the **IStream** interface for other error return values.

Remarks

The **OleDoAutoConvert** function automatically converts an object if automatic conversion has previously been specified in the registry by the **OleSetAutoConvert** function. Object conversion means that the object is permanently associated with a new CLSID. Automatic conversion is typically specified by the setup program for a new version of an object application, so that objects created by its older versions can be automatically updated.

A container application that supports object conversion should call **OleDoAutoConvert** each time it loads an object. If the container uses the **OleLoad** helper function, it need not call **OleDoAutoConvert** explicitly because **OleLoad** calls it internally.

OleDoAutoConvert first determines whether any conversion is required by calling the **OleGetAutoConvert** function, which, if no conversion is required, returns S_OK. If the object requires conversion, **OleDoAutoConvert** modifies and converts the storage object by activating the new object

application. The new object application reads the existing data format, but saves the object in the new native format for the object application.

If the object to be automatically converted is an OLE 1 object, the ItemName string is stored in a stream called "\1Ole10ItemName." If this stream does not exist, the object's item name is NULL.

The storage object must be in the unloaded state when **OleDoAutoConvert** is called.

See Also

OleSetAutoConvert

OleDraw Quick Info

The **OleDraw** helper function can be used to draw objects more easily. You can use it instead of calling **IViewObject::Draw** directly.

```
WINOLEAPI OleDraw(
  IUnknown * pUnk,          //Pointer to the view object to be drawn
  DWORD dwAspect,           //How the object is to be represented
  HDC hdcDraw,              //Device context on which to draw
  LPCRECT lprcBounds        //Pointer to the rectangle in which the object is drawn
);
```

Parameters

pUnk

[in] Pointer to the **IUnknown** interface on the view object that is to be drawn.

dwAspect

[in] How the object is to be represented. Representations include content, an icon, a thumbnail, or a printed document. Valid values are taken from the enumeration **DVASPECT**. See **DVASPECT** for more information.

hdcDraw

[in] Device context on which to draw. Cannot be a metafile device context.

lprcBounds

[in] Pointer to a **RECT** structure specifying the rectangle in which the object should be drawn. This parameter is converted to a **RECTL** structure and passed to **IViewObject::Draw**.

Return Values

This function supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

Object was successfully drawn.

OLE_E_BLANK

No data to draw from.

E_ABORT

The draw operation was aborted.

VIEW_E_DRAW

An error occurred in drawing.

OLE_E_INVALIDRECT

The rectangle is invalid.

DV_E_NOIVIEWOBJECT

The object doesn't support the **IViewObject** interface.

Remarks

The **OleDraw** helper function calls the **QueryInterface** method for the object specified (*pUnk*), asking for an **IViewObject** interface on that object. Then, **OleDraw** converts the **RECT** structure to a **RECTL** structure, and calls **IViewObject::Draw** as follows:

```
lpViewObj->Draw(dwAspect,-1,0,0,0,hdcDraw,&rectl,0,0,0);
```

Do not use **OleDraw** to draw into a metafile because it does not specify the *lprcWBounds* parameter required for drawing into metafiles.

See Also

IViewObject::Draw

OleDuplicateData Quick Info

Duplicates the data found in the specified handle and returns a handle to the duplicated data. The source data is in a clipboard format. Use this function to help implement some of the data transfer interfaces such as **IDataObject**.

```
HANDLE OleDuplicateData(
  HANDLE hSrc,              //Handle of the source data
  CLIPFORMAT cfFormat,      //Clipboard format of the source data
  UINT uiFlags              //Flags used in global memory allocation
);
```

Parameters

hSrc

[in] Handle of the source data.

cfFormat

[in] Clipboard format of the source data.

uiFlags

[in] Flags to be used to allocate global memory for the copied data. These flags are passed to **GlobalAlloc**. If the value of *uiFlags* is NULL, GMEM_MOVEABLE is used as a default flag.

Return Values

handle

When the function is successful, contains the handle to the new data because data was successfully duplicated.

NULL

A NULL return value indicates that there was an error duplicating data.

Remarks

The CF_METAFILEPICT, CF_PALETTE, or CF_BITMAP formats receive special handling. They are GDI handles and a new GDI object must be created instead of just copying the bytes. All other formats are duplicated byte-wise. For the formats that are duplicated byte-wise, *hSrc* must be a global memory handle.

OleFlushClipboard Quick Info

Carries out the clipboard shutdown sequence. It also releases the **IDataObject** pointer that was placed on the clipboard by the **OleSetClipboard** function.

WINOLEAPI OleFlushClipboard();

Return Values

S_OK

The clipboard has been flushed.

CLIPBRD_E_CANT_OPEN

The Windows **OpenClipboard** function used within **OleFlushClipboard** failed.

CLIPBRD_E_CANT_CLOSE

The Windows **CloseClipboard** function used within **OleFlushClipboard** failed.

Remarks

OleFlushClipboard renders the data from a data object onto the clipboard and releases the **IDataObject** pointer to the data object. While the application that put the data object on the clipboard is running, the clipboard holds only a pointer to the data object, thus saving memory. If you are writing an application that acts as the source of a clipboard operation, you can call the **OleFlushClipboard** function when your application is closed, such as when the user exits from your application. Calling **OleFlushClipboard** enables pasting and paste-linking of OLE objects after application shutdown.

Before calling **OleFlushClipboard**, you can easily determine if your data is still on the clipboard with a call to the **OleIsCurrentClipboard** function.

OleFlushClipboard leaves all formats offered by the data transfer object, including the OLE 1 compatibility formats, on the clipboard so they are available after application shutdown. In addition to OLE 1 compatibility formats, these include all formats offered on a global handle medium (all except for TYMED_FILE) and formatted with a NULL target device. For example, if a data-source application offers a particular clipboard format (say cfFOO) on an **IStorage** object, and calls the **OleFlushClipboard** function, the storage object is copied into memory and the *hglobal* memory handle is put on the Clipboard.

To retrieve the information on the clipboard, you can call the **OleGetClipboard** function from another application, which creates a default data object, and the *hglobal* from the clipboard again becomes a storage object. Furthermore, the **FORMATETC** enumerator and the **IDataObject::QueryGetData** method would all correctly indicate that the original clipboard format (cfFOO) is again available on a TYMED_ISTORAGE.

To empty the clipboard, call the **OleSetClipboard** function specifying a NULL value for its parameter. The application should call this when it closes if there is no need to leave data on the clipboard after shutdown, or if data will be placed on the clipboard using the standard Windows clipboard functions.

See Also

OleGetClipboard, **OleSetClipboard**, **OleIsCurrentClipboard**, **IDataObject**

OleGetAutoConvert Quick Info

Determines whether the registry is set for objects of a specified CLSID to be automatically converted to another CLSID, and if so, retrieves the new CLSID.

```
WINOLEAPI OleGetAutoConvert(

  REFCLSID clsidOld,      //CLSID of the object to be converted
  LPCLSID pClsidNew       //Pointer to new CLSID for object being converted
);
```

Parameters

clsidOld

[in] CLSID for an object to determine whether that CLSID is set for automatic conversion.

pClsidNew

[out] Pointrt to where the new CLSID, if any, is written. If auto-conversion for *clsidOld* is not set in the registry, *clsidOld* is written to that location. The *pClsidNew* parameter is never NULL.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

IA value was successfully returned through the *pclsidNew* parameter.

REGDB_E_CLASSNOTREG

The *clsidOld* CLSID is not properly registered in the registry.

REGDB_E_READREGDB

Error reading the registry.

REGDB_E_KEYMISSING

Auto-convert is not active or there was no registry entry for the *clsidOld* parameter.

Remarks

The **OleGetAutoConvert** function returns the AutoConvertTo entry in the registry for the specified object. The AutoConvertTo subkey specifies whether objects of a given CLSID are to be automatically converted to a new CLSID. This is usually used to convert files created by older versions of an application to the current version. If there is no AutoConvertTo entry, this function returns the value of *clsidOld*.

The **OleDoAutoConvert** function calls **OleGetAutoConvert** to determine if the object specified is to be converted. A container application that supports object conversion should call **OleDoAutoConvert** each time it loads an object. If the container uses the **OleLoad** helper function, it need not call **OleDoAutoConvert** explicitly because **OleLoad** calls it internally.

To set up automatic conversion of a given class, you can call the **OleSetAutoConvert** function (typically in the setup program of an application installation). This function uses the AutoConvertTo subkey to tag a class of objects for automatic conversion to a different class of objects. This is a subkey of the CLSID key,

and contains the following information:

```
CLSID\{clsid}=MainUserTypeName\AutoConvertTo = clsid
```

See Also

OleSetAutoConvert, **OleDoAutoConvert**

OleGetClipboard Quick Info

Retrieves a data object that you can use to access the contents of the clipboard.

```
WINOLEAPI OleGetClipboard(
  IDataObject ** ppDataObj      //Indirect pointer to the interface on the data object
);
```

Parameter

ppDataObj

[out] Indirect pointer to the **IDataObject** interface on the clipboard data object.

Return Values

This function supports the standard return values E_INVALIDARG and E_OUTOFMEMORY, as well as the following:

S_OK

The data object was successfully retrieved.

CLIPBRD_E_CANT_CLOSE

The Windows **CloseClipboard** function used within **OleGetClipboard** failed.

CLIPBRD_E_CANT_OPEN

The Windows **OpenClipboard** function used with **OleGetClipboard** failed.

Remarks

If you are writing an application that can accept data from the clipboard, call the **OleGetClipboard** function to get a pointer to the **IDataObject** interface that you can use to retrieve the contents of the clipboard.

OleGetClipboard handles three cases:

1. The application that placed data on the clipboard with the **OleSetClipboard** function is still running.
2. The application that placed data on the clipboard with the **OleSetClipboard** function has subsequently called the **OleFlushClipboard** function.
3. There is data from a non-OLE application on the clipboard.

In the first case, the clipboard data object returned by **OleGetClipboard** may forward calls as necessary to the original data object placed on the clipboard and, thus, can potentially make RPC calls.

In the second case, OLE creates a default data object and returns it to the user. Because the data on the Clipboard originated from an **OleSetClipboard** call, the OLE-provided data object contains more accurate information about the type of data on the Clipboard. In particular, the original medium (TYMED) on which the data was offered is known. Thus, if a data-source application offers a particular clipboard format, for example cfFOO, on a storage object and calls the **OleFlushClipboard** function, the storage object is copied into memory and the *hglobal* memory handle is put on the Clipboard. Then, when the **OleGetClipboard** function creates its default data object, the *hglobal* from the clipboard again becomes

an **IStorage** object. Furthermore, the **FORMATETC** enumerator and the **IDataObject::QueryGetData** method would all correctly indicate that the original clipboard format (cfFOO) is again available on a TYMED_ISTORAGE.

In the third case, OLE still creates a default data object, but there is no special information about the data in the Clipboard formats (particularly for application-defined Clipboard formats). Thus, if an hGlobal-based storage medium were put on the Clipboard directly by a call to the **SetClipboardData** function, the **FORMATETC** enumerator and the **IDataObject::QueryGetData** method would not indicate that the data was available on a storage medium. A call to the **IDataObject::GetData** method for TYMED_ISTORAGE would succeed, however. Because of these limitations, it is strongly recommended that OLE-aware applications interact with the Clipboard using the OLE Clipboard functions.

The clipboard data object created by the **OleGetClipboard** function has a fairly extensive **IDataObject** implementation. The OLE-provided data object can convert OLE 1 clipboard format data into the representation expected by an OLE 2 caller. Also, any structured data is available on any structured or flat medium, and any flat data is available on any flat medium. However, GDI objects (such as metafiles and bitmaps) are only available on their respective mediums.

Note that the *tymed* member of the **FORMATETC** structure used in the **FORMATETC** enumerator contains the union of supported mediums. Applications looking for specific information (such as whether CF_TEXT is available on TYMED_HGLOBAL) should do the appropriate bit masking when checking this value.

If you call the **OleGetClipboard** function, you should only hold on to the returned **IDataObject** for a very short time. It consumes resources in the application that offered it.

See Also

OleSetClipboard

OleGetIconOfClass Quick Info

Returns a handle to a metafile containing an icon and a string label for the specified CLSID.

```
HGLOBAL OleGetIconOfClass(
  REFCLSID rclsid,          //CLSID for which information is requested
  LPOLESTR lpszLabel,       //Pointer to string to use as label for icon
  BOOL fUseTypeAsLabel      //Whether to use CLSID's user type name as icon label
);
```

Parameters

rclsid

[in] CLSID for which the icon and string are requested.

lpszLabel

[in] Pointer to a string to use as a label for the icon.

fUseTypeAsLabel

[in] Whether or not to use the user type string in the CLSID as the icon label.

Return Value

HGLOBAL

The *hGlobal* value returned when the function succeeds is a handle to a metafile that contains an icon and label for the specified CLSID. If the CLSID cannot be found in the registration database, NULL is returned.

See Also

OleGetIconOfFile, **OleMetafilePictFromIconAndLabel**

OleGetIconOfFile Quick Info

Returns a handle to a metafile containing an icon and string label for the specified file name.

```
HGLOBAL OleGetIconOfFile(

  LPOLESTR lpszPath,        //Pointer to string that specifies the file for which info is requested
  BOOL fUseFileAsLabel      //Whether to use the file name as the icon label
);
```

Parameters

lpszPath

[in] Pointer to a file for which the icon and string are requested.

fUseFileAsLabel

[in] Whether or not to use the file name as the icon label.

Return Value

HGLOBAL

The *hGlobal* returned is a handle to a metafile that contains an icon and label for the specified file. If there is no CLSID in the registration database for the file, then the string "Document" is used. If the value of *lpszPath* is NULL, then NULL is returned.

See Also

OleGetIconOfClass, **OleMetafilePictFromIconAndLabel**

OleIconToCursor Quick Info

Converts an icon to a cursor. For Win32 applications, this function calls the Win32 function CopyCursor(hIcon).

```
STDAPI OleIconToCursor(
  HINSTANCE hinstExe,     //Ignored in Win32
  HICON hIcon             //Handle to the icon
);
```

Parameters

hinstExe

[in] Ignored in Win32.

hIcon

[in] Handle to the icon to be converted to a cursor.

Remarks

The return value is an HCURSOR for the new cursor object. The caller is responsible for deleting this cursor with the Win32 function **DestroyCursor**. If the conversion could not be completed, the return value is NULL.

OleInitialize Quick Info

The **OleInitialize** function initializes the OLE library. You must initialize the library before you can call OLE functions.

```
WINOLEAPI OleInitialize(

  LPVOID pvReserved      //Reserved
);
```

Parameter

pvReserved

[in] In 32-bit OLE, reserved; must be NULL. The 32-bit version of OLE does not support applications replacing OLE's allocator and if the parameter is not NULL, **OleInitialize** returns E_INVALIDARG.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED, as well as the following:

S_OK

The library was initialized successfully.

S_FALSE

The OLE library is already initialized; a pointer to the IMalloc implementation was not used.

OLE_E_WRONGCOMPOBJ

Indicates COMPOBJ.DLL is the wrong version for OLE2.DLL.

Remarks

Compound document applications *must* call **OleInitialize** before calling any other function in the OLE library. **OleInitialize** calls the **CoInitialize** function internally.

Typically, **OleInitialize** is called only once in the process that uses the OLE library. There can be multiple calls, but subsequent calls return S_FALSE. To close the library gracefully, each successful call to **OleInitialize**, including those that return S_FALSE, *must* be balanced by a corresponding call to the **OleUninitialize** function.

See Also

OleUninitialize, **CoInitialize**

OleIsCurrentClipboard Quick Info

Determines whether the data object pointer previously placed on the clipboard by the **OleSetClipboard** function is still on the clipboard.

```
WINOLEAPI OleIsCurrentClipboard(

  IDataObject * pDataObject      //Pointer to the data object previously copied or cut
);
```

Parameter

pDataObject

[in] Pointer to the **IDataObject** interface on the data object containing clipboard data of interest, which the caller previously placed on the clipboard.

Return Values

S_OK

The **IDataObject** pointer is currently on the clipboard and the caller is the owner of the clipboard.

S_FALSE

The indicated pointer is not on the clipboard.

Remarks

OleIsCurrentClipboard only works for the data object used in the **OleSetClipboard** function. It cannot be called by the consumer of the data object to determine if the object that was on the clipboard at the previous **OleGetClipboard** call is still on the clipboard.

See Also

OleFlushClipboard, **OleSetClipboard**

OleIsRunning Quick Info

Determines whether a compound document object is currently in the running state.

```
BOOL OleIsRunning(

  LPOLEOBJECT pObject      //Pointer to the interface
);
```

Parameter

pObject

[in] Pointer to the **IOleObject** interface on the object of interest.

Return Value

The return value is TRUE if the object is running; otherwise, it is FALSE.

Remarks

You can use **OleIsRunning** and **IRunnableObject::IsRunning** interchangeably. **OleIsRunning** queries the object for a pointer to the **IRunnableObject** interface and calls its **IsRunning** method. If successful, the function returns the results of the call to **IRunnableObject::IsRunning**.

Note The implementation of **OleIsRunning** in earlier versions of OLE differs from that described here.

See Also

IRunnableObject::IsRunning

OleLoad Quick Info

Loads into memory an object nested within a specified storage object.

```
WINOLEAPI OleLoad(
  IStorage * pStg,              //Pointer to the storage object from which to load
  REFIID riid,                  //Reference to the identifier interface
  IOleClientSite * pClientSite, //Pointer to the client site for the object
  LPVOID * ppvObj               //Indirect pointer to the newly loaded object
);
```

Parameters

pStg

[in] Pointer to the **IStorage** interface on the storage object from which to load the specified object.

riid

[in] Reference to the identifier of the interface that the caller wants to use to communicate with the object once it is loaded.

pClientSite

[in] Pointer to the **IOleClientSite** interface on the client site object being loaded.

ppvObj

[out] When successful, indirect pointer to the interface specified in *riid* on the newly loaded object.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The object was loaded successfully.

E_NOINTERFACE

The object does not support the specified interface.

This function can return any of the error values returned by the **IPersistStorage::Load** method.

Remarks

OLE containers load objects into memory by calling this function. When calling the **OleLoad** function, the container application passes in a pointer to the open storage object in which the nested object is stored. Typically, the nested object to be loaded is a child storage object to the container's root storage object. Using the OLE information stored with the object, the object handler (usually, the default handler) attempts to load the object. On completion of the **OleLoad** function, the object is said to be in the loaded state with its object application not running.

Some applications load all of the object's native data. Containers often defer loading the contained objects until required to do so. For example, until an object is scrolled into view and needs to be drawn, it does not need to be loaded.

The **OleLoad** function performs the following steps:

1. If necessary, performs an automatic conversion of the object (see the **OleDoAutoConvert** function).
2. Gets the CLSID from the open storage object by calling the **IStorage::Stat** method.
3. Calls the **CoCreateInstance** function to create an instance of the handler. If the handler code is not available, the default handler is used (see the **OleCreateDefaultHandler** function).
4. Calls the **IOleObject::SetClientSite** method with the *pClientSite* parameter to inform the object of its client site.
5. Calls the **QueryInterface** method for the **IPersistStorage** interface. If successful, the **IPersistStorage::Load** method is invoked for the object.
6. Queries and returns the interface identified by the *riid* parameter.

See Also

ReadClassStg, **IClassFactory::CreateInstance**, **IPersistStorage::Load**

OleLoadFromStream Quick Info

Loads an object from the stream.

```
WINOLEAPI OleLoadFromStream(
  IStream * pStm,          //Pointer to stream from which object is to be loaded
  REFIID iidInterface,     //Interface identifier
  void ** ppvObj           //Indirect pointer to the newly loaded object
);
```

Parameters

pStm

[in] Pointer to the **IStream** interface on the stream from which the object is to be loaded.

iidInterface

[in] Interface identifier (IID) the caller wants to use to communicate with the object once it is loaded.

ppvObj

[out] On successful return, indirect pointer to the interface requested in *iidInterface* on the newly loaded object.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The object was successfully loaded.

E_NOINTERFACE

The specified interface is not supported.

This function can also return any of the error values returned by the **ReadClassStm** and **CoCreateInstance** functions, and the **IPersistStorage::Load** method.

Remarks

This function can be used to load an object that supports the **IPersistStream** interface. The CLSID of the object must immediately precede the object's data in the stream, which is accomplished by the companion function **OleSaveToStream** (or the operations it wraps, which are described under that topic).

If the CLSID for the stream is CLSID_NULL, the *ppvObj* parameter is set to NULL.

See Also

OleSaveToStream

OleLoadPicture Quick Info

Creates a new picture object and initializes it from the contents of a stream. This is equivalent to calling OleCreatePictureIndirect(NULL, ...) followed by **IPersistStream::Load**.

```
STDAPI OleLoadPicture(
  IStream * pStream,     //Pointer to the stream that contains picture's data
  LONG lSize,            //Number of bytes read from the stream
  BOOL fRunmode,         //The opposite of the initial value of the picture's property
  REFIID riid,           //Reference to the identifier of the interface describing the type of interface pointer to return
  VOID ppvObj            //Indirect pointer to the object
);
```

Parameters

pStream

[in] Pointer to the stream that contains the picture's data.

lSize

[in] Number of bytes that should be read from the stream, or zero if the entire stream should be read.

fRunmode

[in] The opposite of the initial value of the **KeepOriginalFormat** property. If TRUE, **KeepOriginalFormat** is set to FALSE and vice-versa.

riid

[in] Reference to the identifier of the interface describing the type of interface pointer to return in *ppvObj*.

ppvObj

[out] Indirect pointer to the interface identified by *riid* on the storage of the object identified by the moniker. If ppvObj is non-NULL, the implementation must call **IUnknown::AddRef** on the parameter; it is the caller's responsibility to call **IUnknown::Release.** If an error occurs, *ppvObj* is set to NULL.

Return Values

This function supports the standard return values E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The picture was created successfully.

E_POINTER

The address in *pStream* or *ppvObj* is not valid. For example, either may be NULL.

E_NOINTERFACE

The object does not support the interface specified in *riid*.

Remarks

The stream must be in BMP (bitmap), WMF (metafile), or ICO (icon) format. A picture object created using **OleLoadPicture** always has ownership of its internal resources (*fOwn*==TRUE is implied).

See Also

OleCreatePictureIndirect, **PICTDESC**

OleLockRunning Quick Info

Locks an already running object into its running state or unlocks it from its running state.

```
WINOLEAPI OleLockRunning(
  LPUNKNOWN pUnknown,     //Pointer to interface
  BOOL fLock,             //Flag indicating whether object is locked
  BOOL fLastUnlockCloses  //Flag indicating whether to close object
);
```

Parameters

pUnknown

[in] Pointer to the **IUnknown** interface on the object, which the function uses to query for a pointer to **IRunnableObject**.

fLock

[in] TRUE locks the object into its running state. FALSE unlocks the object from its running state.

fLastUnlockCloses

[in] TRUE specifies that if the connection being released is the last external lock on the object, the object should close. FALSE specifies that the object should remain open until closed by the user or another process.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The object was successfully locked or unlocked.

Remarks

The **OleLockRunning** function saves you the trouble of calling the **IRunnableObject::LockRunning** method. You can use **OleLockRunning** and **IRunnableObject::LockRunning** interchangeably. With the IUnknown pointer passed in with the *pUnknown* parameter, **OleLockRunning** queries for an **IRunnableObject** pointer. If successful, it calls **IRunnableObject::LockRunning** and returns the results of the call.

Note The implementation of **OleLockRunning** in earlier versions of OLE differs from that described here.

For more information on using this function, see **IRunnableObject::LockRunning**.

See Also

CoLockObjectExternal, **IRunnableObject::LockRunning**, **OleNoteObjectVisible**

OleMetafilePictFromIconAndLabel Quick Info

Creates a **METAFILEPICT** structure that contains a metafile in which the icon and label are drawn.

```
HGLOBAL OleMetafilePictFromIconAndLabel(
  HICON hIcon,                  //Handle to the icon to be drawn into the metafile
  LPOLESTR lpszLabel,           //Pointer to the string to be used as the icon label
  LPOLESTR lpszSourceFile,      //Pointer to the string that contains the path to the icon file
  UINT iIconIndex               //Index of icon in lpszSourceFile
);
```

Parameters

hIcon

[in] Handle to the icon that is to be drawn into the metafile.

lpszLabel

[in] Pointer to the string to be used as the icon label.

lpszSourceFile

[in] Pointer to the string that contains the path and file name of the icon file. This string can be obtained from the user or from the registration database.

iIconIndex

[in] Index to the icon within the *lpszSourceFile* file.

Return Value

HGLOBAL

An *hGlobal* handle to a **METAFILEPICT** structure containing the icon and label. The metafile uses the MM_ANISOTROPIC mapping mode.

See Also

OleGetIconOfClass, **OleGetIconOfFile**

OleNoteObjectVisible Quick Info

Increments or decrements an external reference that keeps an object in the running state.

```
WINOLEAPI OleNoteObjectVisible(
  LPUNKNOWN pUnknown,     //Pointer to the interface on the object in question
  BOOL fVisible           //Whether object is visible
);
```

Parameters

pUnknown

[in] Pointer to the **IUnknown** interface on the object that is to be locked or unlocked.

fVisible

[in] Whether the object is visible. If TRUE, OLE increments the reference count to hold the object visible and alive regardless of external or internal **IUnknown::AddRef** and **IUnknown::Release** operations, registrations, or revocation. If FALSE, OLE releases its hold (decrements the reference count) and the object can be closed.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

Indicates the object was successfully locked or unlocked.

Remarks

The **OleNoteObjectVisible** function calls the **CoLockObjectExternal** function. It is provided as a separate function to reinforce the need to lock an object when it becomes visible to the user and to release the object when it becomes invisible. This creates a strong lock on behalf of the user to ensure that the object cannot be closed by its container while it is visible.

See Also

CoLockObjectExternal

OleQueryCreateFromData Quick Info

Checks whether a data object has one of the formats that would allow it to become an embedded object through a call to either the **OleCreateFromData** or **OleCreateStaticFromData** function.

```
WINOLEAPI OleQueryCreateFromData(

  IDataObject * pSrcDataObject      //Pointer to the data transfer object to be queried
);
```

Parameter

pSrcDataObject

[in] Pointer to the **IDataObject** interface on the data transfer object to be queried.

Return Values

S_OK

Formats that support embedded-object creation are present.

S_FALSE

No formats are present that support either embedded- or static-object creation.

OLE_S_STATIC

Formats that support static-object creation are present.

Remarks

When an application retrieves a data transfer object through a call to the **OleGetClipboard** function, the application should call **OleQueryCreateFromData** as part of the process of deciding to enable or disable the **Edit/Paste** or **Edit/Paste Special...** commands. It tests for the presence of the following formats in the data object:

CF_EMBEDDEDOBJECT
CF_EMBEDSOURCE
cfFileName
CF_METAFILEPICT
CF_DIB
CF_BITMAP

Determining that the data object has one of these formats does not absolutely guarantee that the object creation will succeed, but is intended to help the process.

If **OleQueryCreateFromData** finds one of the CF_METAFILEPICT, CF_BITMAP, or CF_DIB formats and none of the other formats, it returns OLE_S_STATIC, indicating that you should call the **OleCreateStaticFromData** function to create the embedded object.

If **OleQueryCreateFromData** finds one of the other formats (CF_EMBEDDEDOBJECT, CF_EMBEDSOURCE, or cfFileName), even in combination with the static formats, it returns S_OK, indicating that you should call the **OleCreateFromData** function to create the embedded object.

See Also

OleCreateFromData, **OleCreateStaticFromData**, **OleQueryLinkFromData**

OleQueryLinkFromData Quick Info

The **OleQueryLinkFromData** function determines whether an OLE linked object (rather than an OLE embedded object) can be created from a clipboard data object.

```
WINOLEAPI OleQueryLinkFromData(

  IDataObject * pSrcDataObject      //Pointer to the clipboard data object to be used to create the new object
);
```

Parameter

pSrcDataObject

[in] Pointer to the **IDataObject** interface on the clipboard data object from which the object is to be created.

Return Value

Returns S_OK if the **OleCreateLinkFromData** function can be used to create the linked object; otherwise S_FALSE.

Remarks

The **OleQueryLinkFromData** function is similar to the **OleQueryCreateFromData** function, but determines whether an OLE linked object (rather than an OLE embedded object) can be created from the clipboard data object. If the return value is S_OK, the application can then attempt to create the object with a call to **OleCreateLinkFromData**. A successful return from **OleQueryLinkFromData** does not, however, guarantee the successful creation of a link.

OleRegGetUserType Quick Info

Gets the user type of the specified class from the registry. Developers of custom DLL object applications use this function to emulate the behavior of the OLE default handler.

```
WINOLEAPI OleRegGetUserType(

  REFCLSID clsid,              //Class identifier
  DWORD dwFormOfType,          //Specifies form of type name
  LPOLESTR * pszUserType       //Pointer to storage of string pointer
);
```

Parameters

clsid

[in] Class identifier for which user type is requested.

dwFormOfType

[in] Value that describes the form of the user-presentable string from the enumeration **USERCLASSTYPE**.

pszUserType

[out] Pointer to a string that stores the user type on return.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The user type was returned successfully.

REGDB_E_CLASSNOTREG

There is no CLSID registered for the class object.

REGDB_E_READREGDB

There was an error reading the registry.

OLE_E_REGDB_KEY

The *ProgID* = *MainUserTypeName* and *CLSID* = *MainUserTypeName* keys are missing from the registry.

Remarks

Object applications can ask OLE to get the user type name of a specified class in one of two ways. One way is to call **OleRegGetUserType**. The other is to return OLE_S_USEREG in response to calls by the default object handler to **IOleObject::GetUserType**. OLE_S_USEREG instructs the default handler to call **OleRegGetUserType**. Because DLL object applications cannot return OLE_S_USEREG, they must call **OleRegGetUserType,** rather than delegating the job to the object handler.

The **OleRegGetUserType** function and its sibling functions, **OleRegGetMiscStatus**, **OleRegEnumFormatEtc**, and **OleRegEnumVerbs**, provide a way for developers of custom DLL object applications to emulate the behavior of OLE's default object handler in getting information about objects

from the registry. By using these functions, you avoid the considerable work of writing your own, and the pitfalls inherent in working directly in the registry. In addition, you get future enhancements and optimizations of these functions without having to code them yourself.

See Also

IOleObject::GetUserType, **OleRegGetMiscStatus**, **OleRegEnumFormatEtc**, **OleRegEnumVerbs**, **USERCLASSTYPE**

OleRegGetMiscStatus Quick Info

Gets miscellaneous information about the presentation and behaviors supported by the specified CLSID from the registry. Used by developers of custom DLL object applications to emulate the behavior of the OLE default handler.

```
WINOLEAPI OleRegGetMiscStatus(
  REFCLSID clsid,          //Class identifier
  DWORD dwAspect,          //Value specifying aspect of requested class
  DWORD * pdwStatus        //Pointer to returned status information
);
```

Parameters

clsid

[in] CLSID of the class for which status information is requested.

dwAspect

[in] DWORD specifying the presentation aspect of the class for which information is requested. Values are taken from the **DVASPECT** enumeration.

pdwStatus

[out] Pointer to the location of the status information on the function's return.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The status information was returned successfully.

REGDB_E_CLASSNOTREG

No CLSID is registered for the class object.

REGDB_E_READREGDB

An error occurred reading the registry.

OLE_E_REGDB_KEY

The GetMiscStatus key is missing from the registry.

Remarks

Object applications can ask OLE to get miscellaneous status information in one of two ways. One way is to call **OleRegGetMiscStatus.** The other is to return OLE_S_USEREG in response to calls by the default object handler to **IOleObject::GetMiscStatus**. OLE_S_USEREG instructs the default handler to call **OleRegGetMiscStatus**. Because DLL object applications cannot return OLE_S_USEREG, they must call **OleRegGetMiscStatus** rather than delegating the job to the object handler.

OleRegGetMiscStatus and its sibling functions, **OleRegGetUserType**, **OleRegEnumFormatEtc**, and **OleRegEnumVerbs**, provide a way for developers of custom DLL object applications to emulate the behavior of OLE's default object handler in getting information about objects from the registry. By using

these functions, you avoid the considerable work of writing your own, and the pitfalls inherent in working directly in the registry. In addition, you get future enhancements and optimizations of these functions without having to code them yourself.

See Also

DVASPECT, **FORMATETC**, **OLEMISC**, **IOleObject::GetMiscStatus**, **OleRegEnumFormatEtc**, **OleRegEnumVerbs**, **OleRegGetUserType**

OleRegEnumFormatEtc Quick Info

Supplies a pointer to an enumeration object that can be used to enumerate data formats that an OLE object server has registered in the system registry. An object application or object handler calls this function when it must enumerate those formats. Developers of custom DLL object applications use this function to emulate the behavior of the default object handler.

```
WINOLEAPI OleRegEnumFormatEtc(

  REFCLSID clsid,                          //Class identifier
  DWORD dwDirection,                       //Value specifying data formats
  LPENUMFORMATETC * ppenumFormatetc        //Indirect pointer to returned format information
);
```

Parameters

clsid

[in] CLSID of the class whose formats are being requested.

dwDirection

[in] Whether to enumerate formats that can be passed to **IDataObject::GetData** or formats that can be passed to **IDataObject::SetData**. Valid values are taken from the enumeration **DATADIR**.

ppenumFormatetc

[out] Indirect pointer to the **IEnumFORMATETC** interface on the new enumeration object.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The enumerator was returned successfully.

REGDB_E_CLASSNOTREG

There is no CLSID registered for the class object.

REGDB_E_READREGDB

There was an error reading the registry.

OLE_E_REGDB_KEY

The DataFormats/GetSet key is missing from the registry.

Remarks

Object applications can ask OLE to create an enumeration object for FORMATETC structures to enumerate supported data formats in one of two ways. One way is to call **OleRegEnumFormatEtc.** The other is to return OLE_S_USEREG in response to calls by the default object handler to **IDataObject::EnumFormatEtc**. OLE_S_USEREG instructs the default handler to call **OleRegEnumFormatEtc**. Because DLL object applications cannot return OLE_S_USEREG, they must call **OleRegEnumFormatEtc** rather than delegating the job to the object handler. With the supplied **IEnumFORMATETC** pointer to the object, you can call the standard enumeration object methods to do the enumeration.

The **OleRegEnumFormatEtc** function and its sibling functions, **OleRegGetUserType**, **OleRegGetMiscStatus**, and **OleRegEnumVerbs**, provide a way for developers of custom DLL object applications to emulate the behavior of OLE's default object handler in getting information about objects from the registry. By using these functions, you avoid the considerable work of writing your own, and the pitfalls inherent in working directly in the registry. In addition, you get future enhancements and optimizations of these functions without having to code them yourself.

See Also

IDataObject::EnumFormatEtc, **IEnumFORMATETC**

OleRegEnumVerbs Quick Info

Supplies an enumeration of the registered verbs for the specified class. Developers of custom DLL object applications use this function to emulate the behavior of the default object handler.

```
WINOLEAPI OleRegEnumVerbs(

  REFCLSID clsid,                    //Class identifier
  LPENUMOLEVERB * ppenumOleVerb      //Indirect pointer to returned enumerator
);
```

Parameters

clsid

[in] Class identifier whose verbs are being requested.

ppenumOleVerb

[out] On successful return, indirect pointer to an **IEnumOLEVERB** interface on the new enumeration object.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The enumerator was created successfully.

OLEOBJ_E_NOVERBS

No verbs are registered for the class.

REGDB_E_CLASSNOTREG

No CLSID is registered for the class object.

REGDB_E_READREGDB

An error occurred reading the registry.

OLE_E_REGDB_KEY

The DataFormats/GetSet key is missing from the registry.

Remarks

Object applications can ask OLE to create an enumeration object for OLEVERB structures to enumerate supported verbs in one of two ways. One way is to call **OleRegEnumVerbs**. The other way is to return OLE_S_USEREG in response to calls by the default object handler to **IOleObject::EnumVerbs**. OLE_S_USEREG instructs the default handler to call **OleRegEnumVerbs**. Because DLL object applications cannot return OLE_S_USEREG, they must call **OleRegEnumVerbs** rather than delegating the job to the object handler. With the supplied **IEnumOLEVERB** pointer to the object, you can call the standard enumeration object methods to do the enumeration.

The **OleRegEnumVerbs** function and its sibling functions, **OleRegGetUserType**, **OleRegGetMiscStatus**, and **OleRegEnumFormatEtc**, provide a way for developers of custom DLL object applications to emulate the behavior of OLE's default object handler in getting information about

objects from the registry. By using these functions, you avoid the considerable work of writing your own, and the pitfalls inherent in working directly in the registry. In addition, you get future enhancements and optimizations of these functions without having to code them yourself.

See Also

IOleObject::EnumVerbs, **IEnumOLEVERB**

OleRun Quick Info

Puts an OLE compound document object into the running state.

```
WINOLEAPI OleRun(

  LPUNKNOWN pUnknown      //Pointer to interface on the object
);
```

Parameter

pUnknown

[in] Pointer to the **IUnknown** interface on the object, with which it will query for a pointer to the **IRunnableObject** interface, and then call its **Run** method.

Return Values

This function supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The object was successfully placed in the running state.

OLE_E_CLASSDIFF

The source of an OLE link has been converted to a different class.

Remarks

The **OleRun** function puts an object in the running state. The implementation of **OleRun** was changed in OLE 2.01 to coincide with the publication of the **IRunnableObject** interface. You can use **OleRun** and **IRunnableObject::Run** interchangeably. **OleRun** queries the object for a pointer to **IRunnableObject**. If successful, the function returns the results of calling the **IRunnableObject::Run** method.

Note The implementation of **OleRun** in earlier versions of OLE differs from that described here.

For more information on using this function, see **IRunnableObject::Run**.

See Also

IOleLink::BindToSource, **IRunnableObject::Run**

OleSave Quick Info

Saves an object opened in transacted mode into the specified storage object.

```
WINOLEAPI OleSave(
  IPersistStorage * pPS,      //Pointer to the object to be saved
  IStorage * pStg,            //Pointer to the destination storage to which pPS is saved
  BOOL fSameAsLoad            //Whether the object was loaded from pstg or not
);
```

Parameters

pPS

[in] Pointer to the **IPersistStorage** interface on the object to be saved.

pStg

[in] Pointer to the **IStorage** interface on the destination storage object to which the object indicated in *pPS* is to be saved.

fSameAsLoad

[in] TRUE indicates that *pStg* is the same storage object from which the object was loaded or created; FALSE indicates that *pstg* was loaded or created from a different storage object.

Return Values

S_OK

The object was successfully saved.

STG_E_MEDIUMFULL

The object could not be saved due to lack of disk space.

This function can also return any of the error values returned by the **IPersistStorage::Save** method.

Remarks

The **OleSave** helper function handles the common situation in which an object is open in transacted mode and is then to be saved into the specified storage object which uses the OLE-provided compound file implementation. Transacted mode means that changes to the object are buffered until either of the **IStorage** methods **Commit** or **Revert** is called. Callers can handle other situations by calling the **IPersistStorage** and **IStorage** interfaces directly.

OleSave does the following:

1. Calls the **IPersistStorage::GetClassID** method to get the CLSID of the object.
2. Writes the CLSID to the storage object using the **WriteClassStg** function.
3. Calls the **IPersistStorage::Save** method to save the object.
4. If there were no errors on the save; calls the **IStorage::Commit** method to commit the changes.

Note Static objects are saved into a stream called CONTENTS. Static metafile objects get saved in

"placeable metafile format" and static DIB data gets saved in "DIB file format." These formats are defined to be the OLE standards for metafile and DIB. All data transferred using an **IStream** interface or a file (that is, via **IDataObject::GetDataHere**) must be in these formats. Also, all objects whose default file format is a metafile or DIB must write their data into a CONTENTS stream using these standard formats.

See Also

IStorage, **IPersistStorage**

OleSaveToStream Quick Info

Saves an object with the **IPersistStream** interface on it to the specified stream.

```
WINOLEAPI OleSaveToStream(

  IPersistStream * pPStm,     //Pointer to the interface on the object to be saved
  IStream * pStm              //Pointer to the destination stream to which the object is saved
);
```

Parameters

pPStm

[in] Pointer to the **IPersistStream** interface on the object to be saved to the stream. Can be NULL, which has the effect of writing CLSID_NULL to the stream.

pStm

[in] Pointer to the **IStream** interface on the stream in which the object is to be saved.

Return Values

S_OK

The object was successfully saved.

STG_E_MEDIUMFULL

There is no space left on device to save the object.

This function can also return any of the error values returned by the **WriteClassStm** function or the **IPersistStream::Save** method.

Remarks

This function simplifies saving an object that implements the **IPersistStream** interface to a stream. In this stream, the object's CLSID precedes its data. When the stream is retrieved, the CLSID permits the proper code to be associated with the data. The **OleSaveToStream** function does the following:

1. Calls the **IPersistStream::GetClassID** method to get the object's CLSID.
2. Writes the CLSID to the stream with the **WriteClassStm** function.
3. Calls the **IPersistStream::Save** method with *fClearDirty* set to TRUE, which clears the dirty bit in the object.

The companion helper, **OleLoadFromStream**, loads objects saved in this way.

See Also

OleLoadFromStream, **IPersistStream**, **IStream**

OleSetAutoConvert Quick Info

Specifies a CLSID for automatic conversion to a different class when an object of that class is loaded.

```
WINOLEAPI OleSetAutoConvert(

  REFCLSID clsidOld,     //CLSID to be converted
  REFCLSID clsidNew      //New CLSID after conversion
);
```

Parameters

clsidOld

[in] CLSID of the object class to be converted.

clsidNew

[in] CLSID of the object class that should replace *clsidOld*. This new CLSID replaces any existing auto-conversion information in the registry for *clsidOld*. If this value is CLSID_NULL, any existing auto-conversion information for *clsidOld* is removed from the registry.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The object was tagged successfully.

REGDB_E_CLASSNOTREG

The CLSID is not properly registered in the registry.

REGDB_E_READREGDB

Error reading from the registry.

REGDB_E_WRITEREGDB

Error writing to the registry.

REGDB_E_KEYMISSING

Cannot read a key from the registry.

Remarks

The **OleSetAutoConvert** function goes to the system registry, finds the AutoConvertTo subkey under the CLSID specified by *clsidOld*, and sets it to *clsidNew*. This function does not validate whether an appropriate registry entry for *clsidNew* currently exists. These entries appear in the registry as subkeys of the CLSID key:

```
CLSID\{clsid}=MainUserTypeName\AutoConvertTo = clsid
```

Object conversion means that the object's data is permanently associated with a new CLSID. Automatic conversion is typically specified in the setup program of a new version of an object application, so objects created by its older versions can be automatically updated to the new version.

For example, it may be necessary to convert spreadsheets that were created with earlier versions of a spreadsheet application to the new version. The spreadsheet objects from earlier versions have different CLSIDs than the new version. For each earlier version that you want automatically updated, you would call **OleSetAutoConvert** in the setup program, specifying the CLSID of the old version, and that of the new one. Then, whenever a user loads an object from a previous version, it would be automatically updated. To support automatic conversion of obects, a server that supports conversion must be prepared to manually convert objects that have the format of an earlier version of the server. Automatic conversion relies internally on this manual-conversion support.

Before setting the desired **AutoConvertTo** value, setup programs should also call **OleSetAutoConvert** to remove any existing conversion for the new class, by specifying the new class as the *clsidOld* parameter, and setting the *clsidNew* parameter to CLSID_NULL.

See Also

OleDoAutoConvert

OleSetClipboard Quick Info

Places a pointer to a specific data object onto the clipboard. This makes the data object accessible to the **OleGetClipboard** function.

```
WINOLEAPI OleSetClipboard(

  IDataObject * pDataObj      //Pointer to the data object being copied or cut
);
```

Parameter

pDataObj

[in] Pointer to the **IDataObject** interface on the data object from which the data to be placed on the clipboard can be obtained. This parameter can be NULL; in which case the clipboard is emptied.

Return Values

S_OK

The **IDataObject** pointer was placed on the clipboard.

CLIPBRD_E_CANT_OPEN

The Windows **OpenClipboard** function used within **OleSetClipboard** failed.

CLIPBRD_E_CANT_EMPTY

The Windows **EmptyClipboard** function used within **OleSetClipboard** failed.

CLIPBRD_E_CANT_CLOSE

The Windows **CloseClipboard** function used within **OleSetClipboard** failed.

Remarks

If you are writing an application that can act as the source of a clipboard operation, you must do the following:

1. Create a data object (on which is the **IDataObject** interface) for the data being copied or cut to the clipboard. This object should be the same object used in OLE drag-and-drop operations.
2. Call **OleSetClipboard** to place the **IDataObject** pointer onto the clipboard, so it is accessible to the **OleGetClipboard** function. **OleSetClipboard** also calls the **IUnknown::AddRef** method on your data object.
3. If you wish, release the data object once you have placed it on the clipboard to free the **IUnknown::AddRef** counter in your application.
4. If the user is cutting data (deleting it from the document and putting it on to the clipboard), remove the data from the document.

All formats are offered on the clipboard using delayed rendering (the clipboard contains only a pointer to the data object unless a call to **OleFlushClipboard** renders the data onto the clipboard). The formats necessary for OLE 1 compatibility are synthesized from the OLE 2 formats that are present and are also put on the clipboard.

The **OleSetClipboard** function assigns ownership of the clipboard to an internal OLE window handle. The

reference count of the data object is increased by 1, to enable delayed rendering. The reference count is decreased by a call to the **OleFlushClipboard** function or by a subsequent call to **OleSetClipboard** specifying NULL as the parameter value (which clears the clipboard).

When an application opens the clipboard (either directly or indirectly by calling the Win32 **OpenClipboard** function), the clipboard cannot be used by any other application until it is closed. If the clipboard is currently open by another application, **OleSetClipboard** fails. The internal OLE window handle satisfies WM_RENDERFORMAT messages by delegating them to the **IDataObject** implementation on the data object that is on the clipboard.

Specifying NULL as the parameter value for **OleSetClipboard** empties the current clipboard. If the contents of the clipboard are the result of a previous **OleSetClipboard** call and the clipboard has been released, the **IDataObject** pointer that was passed to the previous call is released. The clipboard owner should use this as a signal that the data it previously offered is no longer on the clipboard.

If you need to leave the data on the clipboard after your application is closed, you should call **OleFlushClipboard** instead of **OleSetClipboard**.

See Also

OleFlushClipboard, **OleGetClipboard**, **OleIsCurrentClipboard**

OleSetContainedObject Quick Info

Notifies an object that it is embedded in an OLE container, which ensures that reference counting is done correctly for containers that support links to embedded objects.

```
WINOLEAPI OleSetContainedObject(

  LPUNKNOWN pUnk,      //Pointer to the interface on the embedded object
  BOOL fContained      //Indicates if the object is embedded
);
```

Parameters

pUnk

[in] Pointer to the **IUnknown** interface of the object.

fContained

[in] TRUE if the object is an embedded object; FALSE otherwise.

Return Values

This function supports the standard return values E_INVALIDARG, E_OUTOFMEMORY and E_UNEXPECTED, as well as the following:

S_OK

The object was notified successfully.

Remarks

The **OleSetContainedObject** function notifies an object that it is embedded in an OLE container. The implementation of **OleSetContainedObject** was changed in OLE 2.01 to coincide with the publication of the **IRunnableObject** interface. You can use **OleSetContainedObject** and the **IRunnableObject::SetContainedObject** method interchangeably. The **OleSetContainedObject** function queries the object for a pointer to the **IRunnableObject** interface. If successful, the function returns the results of calling **IRunnableObject::SetContainedObject**.

Note The implementation of **OleSetContainedObject** in earlier versions of OLE differs from that described here.

See Also

IRunnableObject::SetContainedObject

OleSetMenuDescriptor Quick Info

Installs or removes OLE dispatching code from the container's frame window.

```
WINOLEAPI OleSetMenuDescriptor(
  HOLEMENU holemenu,                    //Handle to the composite menu descriptor
  HWND hwndFrame,                       //Handle to the container's frame window
  HWND hwndActiveObject,                //Handle to the object's in-place activation window
  LPOLEINPLACEFRAME lpFrame,            //Pointer to the container's frame window
  LPOLEINPLACEACTIVEOBJECT lpActiveObj  //Active in-place object
);
```

Parameters

holemenu

[in] Handle to the composite menu descriptor returned by the **OleCreateMenuDescriptor** function. If NULL, the dispatching code is unhooked.

hwndFrame

[in] Handle to the container's frame window where the in-place composite menu is to be installed.

hwndActiveObject

[in] Handle to the object's in-place activation window. OLE dispatches menu messages and commands to this window.

lpFrame

[in] Pointer to the **IOleInPlaceFrame** interface on the container's frame window.

lpActiveObj

[in] Pointer to the **IOleInPlaceActiveObject** interface on the active in-place object.

Return Values

This function supports the standard return values E_FAIL, E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The menu was installed correctly.

Remarks

The container should call **OleSetMenuDescriptor** to install the dispatching code on *hwndFrame* when the object calls the **IOleInPlaceFrame::SetMenu** method, or to remove the dispatching code by passing NULL as the value for *holemenu* to **OleSetMenuDescriptor**.

If both the *lpFrame* and *lpActiveObj* parameters are non-NULL, OLE installs the context-sensitive help F1 message filter for the application. Otherwise, the application must supply its own message filter.

See Also

OleCreateMenuDescriptor, **IOleInPlaceFrame::SetMenu**, **IOleInPlaceActiveObject**

OleTranslateAccelerator Quick Info

Called by the object application, allows an object's container to translate accelerators according to the container's accelerator table.

```
WINOLEAPI OleTranslateAccelerator(
  LPOLEINPLACEFRAME lpFrame,          //Pointer to send keystrokes
  LPOLEINPLACEFRAMEINFO lpFrameInfo,  //Pointer to accelerator table obtained from container
  LPMSG lpmsg                         //Pointer to structure containing the keystroke
);
```

Parameters

lpFrame

[in] Pointer to the **IOleInPlaceFrame** interface to which the keystroke might be sent.

lpFrameInfo

[in] Pointer to an **OLEINPLACEFRAMEINFO** structure containing the accelerator table obtained from the container.

lpmsg

[in] Pointer to an **MSG** structure containing the keystroke.

Return Values

This function supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The keystroke was processed.

S_FALSE

The object should continue processing this message.

Remarks

Object servers call **OleTranslateAccelerator** to allow the object's container to translate accelerator keystrokes according to the container's accelerator table, pointed to by *lpFrameInfo*. While a contained object is the active object, the object's server *always* has first chance at translating any messages received. If this is not desired, the server calls **OleTranslateAccelerator** to give the object's container a chance. If the keyboard input matches an accelerator found in the container-provided accelerator table, **OleTranslateAccelerator** passes the message and its command identifier on to the container through the **IOleInPlaceFrame::TranslateAccelerator** method. This method returns S_OK if the keystroke is consumed; otherwise it returns S_FALSE.

The **OleTranslateAccelerator** function is intended to be called only by local server applications and not by in-process servers. For objects managed by local servers, keyboard input goes directly to the server's message pump. If the object does not translate the key, **OleTranslateAccelerator** gives the container application an opportunity to do it. For objects managed by in-process servers, the keyboard input goes directly to the container's message pump.

Note Accelerator tables for containers should be defined so they will work properly with object applications that do their own accelerator keystroke translations. These tables should take the form:

```
"char", wID, VIRTKEY, CONTROL
```

This is the most common way to describe keyboard accelerators. Failure to do so can result in keystrokes being lost or sent to the wrong object during an in-place session.

Objects can call the **IsAccelerator** function to see whether the accelerator keystroke belongs to the object or the container.

See Also

IsAccelerator, **IOleInPlaceFrame::TranslateAccelerator**

OleTranslateColor Quick Info

Converts an **OLE_COLOR** type to a **COLORREF**.

```
STDAPI OleTranslateColor (
  OLE_COLOR clr,          //Color to be converted into a COLORREF
  HPALETTE hpal,          //Palette used for conversion
  COLORREF *pcolorref     //Pointer to the caller's variable that receives the converted result
);
```

Parameters

clr

[in] The OLE color to be converted into a **COLORREF**.

hpal

[in] Palette used as a basis for the conversion.

pcolorref

[out] Pointer to the caller's variable that receives the converted **COLORREF** result. This can be NULL, indicating that the caller wants only to verify that a converted color exists.

Return Values

This function supports the standard return values E_INVALIDARG and E_UNEXPECTED, as well as the following:

S_OK

The color was translated successfully.

Remarks

The following table describes the color conversion:

OLE_COLOR	***hPal***	**Resulting COLORREF**
invalid		Undefined (E_INVALIDARG)
0x800000*xx*, *xx* is not a valid Win32 GetSysColor index		Undefined (E_INVALIDARG)
	invalid	Undefined (E_INVALIDARG)
0x0100*iiii*, *iiii* is not a valid palette index	valid palette	Undefined (E_INVALIDARG)
0x800000*xx*, *xx* is a valid GetSysColor index	NULL	0x00*bbggrr*
0x0100*iiii*, *iiii* is a valid palette index	NULL	0x0100*iiii*
0x02*bbggrr* (palette relative)	NULL	0x02*bbggrr*

0x00*bbggrr*	NULL	0x00*bbggrr*
0x800000*xx*, *xx* is a valid GetSysColor index	valid palette	0x00*bbggrr*
0x0100*iiii*, *iiii* is a valid palette index in *hPal*	valid palette	0x0100*iiii*
0x02*bbggrr* (palette relative)	valid palette	0x02*bbggrr*
0x00*bbggrr*	valid palette	0x02*bbggrr*

OleUIAddVerbMenu Quick Info

Adds the Verb menu for the specified object to the given menu.

```
BOOL OleUIAddVerbMenu(

  LPOLEOBJECT *lpOleObj,        //Pointer to the object
  LPCTSTR lpszShortType,        //Pointer to the short name corresponding to the object
  HMENU hMenu,                  //Handle to the menu to modify
  UINT uPos,                    //Position of the menu item.
  UINT uIDVerbMin,              //Value at which to start the verbs
  UINT uIDVerbMax,              //Maximum identifier value for object verbs
  BOOL bAddConvert,             //Whether to add convert item
  UINT idConvert,               //Value to use for the convert item
  HMENU FAR * lphMenu           //Pointer to the cascading verb menu, if created
);
```

Parameters

lpOleObj

[in] Pointer to the **IOleObject** interface on the selected object. If this is NULL, then a default disabled menu item is created.

lpszShortType

[in] Pointer to the short name defined in the registry (AuxName==2) for the object identified with *lpOleObj*. If the string is NOT known, then NULL may be passed. If NULL is passed, **IOleObject::GetUserType** is called to retrieve it. If the caller has easy access to the string, it is faster to pass it in.

hMenu

[in] Handle to the menu in which to make modifications.

uPos

[in] Position of the menu item.

iIDVerbMin

[in] The **UINT** identifier value at which to start the verbs.

uIDVerbMax

[in] The **UINT** Maximum identifier value to be used for object verbs. If *uIDVerbMax* is 0, then no maximum identifier value is used.

bAddConvert

[in] The BOOL specifying whether or not to add a Convert item to the bottom of the menu (preceded by a separator).

idConvert

[in] The **UINT** identifier value to use for the Convert menu item, if *bAddConvert* is TRUE.

lphMenu

[out] An HMENU pointer to the cascading verb menu if it's created. If there is only one verb, this will be filled with NULL.

Return Values

TRUE

Indicates *lpOleObj* was valid and at least one verb was added to the menu.

FALSE

Indicates *lpOleObj* was NULL and a disabled default menu item was created.

Remarks

If the object has one verb, the verb is added directly to the given menu. If the object has multiple verbs, a cascading sub-menu is created.

OleUIBusy Quick Info

Invokes the standard Busy dialog box, allowing the user to manage concurrency.

```
UINT OleUIBusy(
  LPOLEUIBUSY lpBZ     //Pointer to the initialization structure
);
```

Parameter

lpBZ

[in] Pointer to an **OLEUIBUSY** structure that contains information used to initialize the dialog box.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user has pressed the Cancel button and that the caller should cancel the operation.

OLEUI_BZ_SWITCHTOSELECTED

The user has pressed Switch To and **OleUIBusy** was unable to determine how to switch to the blocking application. In this case, the caller should either take measures to attempt to resolve the conflict itself, if possible, or retry the operation. **OleUIBusy** will only return OLEUI_BZ_SWITCHTOSELECTED if the user has pressed the Switch To button, *hTask* is NULL and the BZ_NOTRESPONDING flag is set.

OLEUI_BZ_RETRYSELECTED

The user has either pressed the Retry button or attempted to resolve the conflict (probably by switching to the blocking application). In this case, the caller should retry the operation.

OLEUI_BZ_CALLUNBLOCKED

The dialog box has been informed that the operation is no longer blocked.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_BZERR_HTASKINVALID

The *hTask* specified in the *hTask* member of the **OLEUIBUSY** structure is invalid.

Remarks

The standard OLE Server Busy dialog box notifies the user that the server application is not receiving messages. The dialog box then asks the user to cancel the operation, switch to the task that is blocked, or continue waiting.

See Also

OLEUIBUSY

OleUICanConvertOrActivateAs Quick Info

Determines if there are any OLE object classes in the registry that can be used to convert or activate the specified CLSID from.

```
BOOL OleUICanConvertOrActivateAs(

  REFCLSID rClsid,          //CLSID of the specified class
  BOOL fIsLinkedObject,     //Whether the original object was a linked object
  WORD wFormat              //Format of the original class
);
```

Parameters

rClsid

[in] The CLSID of the class for which the information is required.

fIsLinkedObject

[in] TRUE if the original object is a linked object; FALSE otherwise.

wFormat

[in] Format of the original class.

Return Values

TRUE

The specified class can be converted to or activated as another class.

FALSE

The specified class cannot be converted to or activated as another class.

Remarks

OleUICanConvertOrActivateAs searches the registry for classes that include *wFormat* in their \Conversion\Readable\Main, \Conversion\ReadWriteable\Main, and \DataFormats\DefaultFile entries.

This function is useful for determining if a Convert... menu item should be disabled. If the CF_DISABLEDISPLAYASICON flag is specified in the call to **OleUIConvert**, then the Convert... menu item should be enabled only if **OleUICanConvertOrActivateAs** returns TRUE.

See Also

OleUIConvert

OleUIChangeIcon Quick Info

Invokes the standard Change Icon dialog box, which allows the user to select an icon from an icon file, executable, or DLL.

```
UINT OleUIChangeIcon(

  LPOLEUICHANGEICON lpCI      //Pointer to the in-out structure for this dialog box
);
```

Parameter

lpCI

[in] Pointer to the in-out **OLEUICHANGEICON** structure for this dialog box.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_CIERR_MUSTHAVECLSID

The *clsid* member was not the current CLSID.

OLEUI_CIERR_MUSTHAVECURRENTMETAFILE

The *hMetaPict* member was not the current metafile.

OLEUI_CIERR_SZICONEXEINVALID

The *szIconExe* value was invalid.

Remarks

OleUIChangeIcon uses information contained in the **OLEUICHANGEICON** structure.

See Also

OLEUICHANGEICON

OleUIChangeSource Quick Info

Invokes the Change Source dialog box, allowing the user to change the source of a link.

```
UINT OleUIChangeSource(

  LPOLEUICHANGESOURCE lpCS      //Pointer to the in-out structure
);
```

Parameter

lpCS

[in] Pointer to the in-out **OLEUICHANGESOURCE** structure for this dialog box.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_CSERR_LINKCNTRNULL

The *lpOleUILinkContainer* value is NULL.

OLEUI_CSERR_LINKCNTRINVALID

The *lpOleUILinkContainer* value is invalid.

OLEUI_CSERR_FROMNOTNULL

The *lpszFrom* value is not NULL.

OLEUI_CSERR_TONOTNULL

The *lpszTo* value is not NULL.

OLEUI_CSERR_SOURCEINVALID

The *lpszDisplayName* or *nFileLength* value is invalid, or cannot retrieve the link source.

OLEUI_CSERR_SOURCEPARSEERROR

The *nFilename* value is wrong.

Remarks

The link source is not changed by the Change Source dialog box itself. Instead, it is up to the caller to change the link source using the returned file and item strings. The Edit Links dialog box typically does this for the caller.

See Also

OLEUICHANGESOURCE, **OleUIEditLinks**, **IOleUILinkContainer**

OleUIConvert Quick Info

Invokes the standard Convert dialog box, allowing the user to change the type of a single specified object, or the type of all OLE objects of the specified object's class.

```
UINT OleUIConvert(

  LPOLEUICONVERT lpCV      //Pointer to initialization structure
);
```

Parameter

lpCV

[in] Pointer to an **OLEUICONVERT** structure that contains information used to initialize the dialog box.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_CTERR_CLASSIDINVALID

A *clsid* value was invalid.

OLEUI_CTERR_DVASPECTINVALID

The *dvAspect* value was invalid. This member specifies the aspect of the object.

OLEUI_CTERR_CBFORMATINVALID

The *wFormat* value was invalid. This member specifies the data format of the object.

OLEUI_CTERR_STRINGINVALID

A string value (for example, *lpszUserType* or *lpszDefLabel*) was invalid.

Remarks

OleUIConvert populates the Convert dialog box's list box with object classes by traversing the Registry and looking for entries in the Readable and ReadWritable keys. Every class that includes the original class' default file format in its Readable key is added to the Convert list, and every class that includes the original class' default file format in its ReadWritable key is added to the Activate As list. The Convert list is shown in the dialog box's list box when the Convert radio button is selected (the default selection), and the Activate As list is shown when Activate As is selected.

Note that you can change the type of all objects of a given class only when CF_CONVERTONLY is not specified.

The convert command, which invokes this function, should only be made available to the user if **OleUIConvertOrActivateAs** returns S_OK.

See Also

OLEUICONVERT, **OleUICanConvertOrActivateAs**

OleUIEditLinks Quick Info

Invokes the standard Links dialog box, allowing the user to make modifications to a container's linked objects.

```
UINT OleUIEditLinks(

  LPOLEUIEDITLINKS lpEL      //Pointer to the initialization structure
);
```

Parameter

lpEL

[in] Pointer to an **OLEUIEDITLINKS** structure that contains information used to initialize the dialog box.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

See Also

OLEUIEDITLINKS, **IOleUILinkContainer**

OleUIInsertObject Quick Info

Invokes the standard Insert Object dialog box, which allows the user to select an object source and class name, as well as the option of displaying the object as itself or as an icon.

```
UINT OleUIInsertObject(
  LPOLEUIINSERTOBJECT lpIO      //Pointer to the in-out structure
);
```

Parameter

lpIO

[in] Pointer to the in-out **OLEUIINSERTOBJECT** structure for this dialog box.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_IOERR_LPSZFILEINVALID

The *lpszFile* value is invalid or user has insufficient write access permissions.. This *lpszFile* member points to the name of the file linked to or inserted.

OLEUI_IOERR_LPFORMATETCINVALID

The *lpFormatEtc* value is invalid. This member identifies the desired format.

OLEUI_IOERR_PPVOBJINVALID

The *ppvOjb* value is invalid. This member points to the location where the pointer for the object is returned.

OLEUI_IOERR_LPIOLECLIENTSITEINVALID

The *lpIOleClientSite* value is invalid. This member points to the client site for the object.

OLEUI_IOERR_LPISTORAGEINVALID

The *lpIStorage* value is invalid. This member points to the storage to be used for the object.

OLEUI_IOERR_SCODEHASERROR

The *sc* member of *lpIO* has additional error information.

OLEUI_IOERR_LPCLSIDEXCLUDEINVALID

The *lpClsidExclude* value is invalid. This member contains the list of CLSIDs to exclude.

OLEUI_IOERR_CCHFILEINVALID

The *cchFile* or *lpszFile* value is invalid. The *cchFile* member specifies the size of the *lpszFile* buffer. The *lpszFile* member points to the name of the file linked to or inserted.

Remarks

OleUIInsertObject allows the user to select the type of object to be inserted from a list box containing the object applications registered on the user's system. To populate that list box, **OleUIInsertObject** traverses the registry, adding every object server it finds that meets the following criteria:

- The registry entry does not include the **NotInsertable** key.
- The registry entry includes an OLE 1.0 style **Protocol\\StdFileEditing\\Server** key.
- The registry entry includes the **Insertable** key.
- The object's CLSID is not included in the list of objects to exclude (the *lpClsidExclude* member of **OLEUIINSERTOBJECT**).

By default, **OleUIInsertObject** does not validate object servers, however, if the IOF_VERIFYSERVEREXIST flag is included in the *dwFlags* member of the **OLEUIINSERTOBJECT** structure, **OleUIInsertObject** verifies that the server exists. If it does not exist, then the server's object is not added to the list of available objects. Server validation is a time-extensive operation and is a significant performance factor.

To free an HMETAFILEPICT returned from the Insert Object or Paste Special dialog box, delete the attached metafile on the handle, as follows:

```
void FreeHmetafilepict(HMETAFILEPICT hmfp)
{
    if (hmfp != NULL)
        {
        LPMETAFILEPICT pmfp = GlobalLock(hmfp);

        DeleteMetaFile(pmfp->hMF);
        GlobalUnlock(hmfp);
        GlobalFree(hmfp);
        }
}  // FreeHmetafilepict
```

See Also

OLEUIINSERTOBJECT, **OpenFile**, **DeleteMetaFile**, **GlobalUnlock**, **GlobalFree** in Win32

OleUIObjectProperties Quick Info

Invokes the Object Properties dialog box, which displays General, View, and Link information about an object.

```
UINT OleUIObjectProperties(

  LPOLEUIOBJECTPROPS lpOP      //Pointer to the structure
);
```

Parameter

lpOP

[in] Pointer to the **OLEUIOBJECTPROPS** structure.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_OPERR_SUBPROPNULL

lpGP or *lpVP* is NULL, or *dwFlags* and OPF_OBJECTISLINK and *lpLP* are NULL.

OLEUI_OPERR_SUBPROPINVALID

Insufficient write-access permissions for the structures pointed to by *lpGP*, *lpVP*, or *lpLP*.

OLEUI_OPERR_PROPSHEETNULL

The *lpLP* value is NULL.

OLEUI_OPERR_PROPSHEETINVALID

Insufficient write access for one or more of the structures used by OLEUIOBJECTPROPS.

OLEUI_OPERR_SUPPROP

The sub-link property pointer, *lpLP*, is NULL.

OLEUI_OPERR_PROPSINVALID

Insufficient write access for the sub-link property pointer, *lpLP*.

OLEUI_OPERR_PAGESINCORRECT

Some sub-link properties of the *lpPS* member are incorrect.

OLEUI_OPERR_INVALIDPAGES

Some sub-link properties of the *lpPS* member are incorrect.

OLEUI_OPERR_NOTSUPPORTED

A sub-link property of the *lpPS* member is incorrect.

OLEUI_OPERR_DLGPROCNOTNULL

A sub-link property of the *lpPS* member is incorrect.

OLEUI_OPERR_LPARAMNOTZERO

A sub-link property of the *lpPS* member is incorrect.

OLEUI_GPERR_STRINGINVALID

A string value (for example, *lplpszLabel* or *lplpszType*) is invalid.

OLEUI_GPERR_CLASSIDINVALID

The *clsid* value is invalid.

OLEUI_GPERR_LPCLSIDEXCLUDEINVALID

The *ClsidExcluded* value is invalid.

OLEUI_GPERR_CBFORMATINVALID

The *wFormat* value is invalid.

OLEUI_VPERR_METAPICTINVALID

The *hMetaPict* value is invalid.

OLEUI_VPERR_DVASPECTINVALID

The *dvAspect* value is invalid.

OLEUI_OPERR_PROPERTYSHEET

The *lpPS* value is incorrect.

OLEUI_OPERR_OBJINFOINVALID

The *lpObjInfo* value is NULL or the calling process doesn't have read access.

OLEUI_OPERR_LINKINFOINVALID

The *lpLinkInfo* value is NULL or the calling process doesn't have read access.

Remarks

OleUIObjectProperties is passed an **OLEUIOBJECTPROPS** structure, which supplies the information needed to fill in the General, View, and Link tabs of the Object Properties dialog box.

See Also

IOleUIObjInfo, **IOleUILinkInfo**, **OLEUIOBJECTPROPS**, **OLEUIGNRLPROPS**, **OLEUIVIEWPROPS**, **OLEUILINKPROPS**

OleUIPasteSpecial Quick Info

Invokes the standard Paste Special dialog box, allowing the user to select the format of the clipboard object to be pasted or paste-linked.

```
UINT OleUIPasteSpecial(*
  LPOLEUIPASTESPECIAL lpPS      //Pointer to the in-out structure for this dialog box
);
```

Parameter

lpPS

[in] Pointer to the in-out **OLEUIPASTESPECIAL** structure for this dialog box.

Return Values

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_IOERR_SRCDATAOBJECTINVALID

The *lpSrcDataObject* field of **OLEUIPASTESPECIAL** is invalid.

OLEUI_IOERR_ARRPASTEENTRIESINVALID

The *arrPasteEntries* field of **OLEUIPASTESPECIAL** is invalid.

OLEUI_IOERR_ARRLINKTYPESINVALID

The *arrLinkTypes* field of **OLEUIPASTESPECIAL** is invalid.

OLEUI_PSERR_CLIPBOARDCHANGED

The clipboard contents changed while the dialog box was displayed.

OLEUI_PSERR_GETCLIPBOAARDFAILED

The *lpSrcDataObj* member is incorrect.

Remarks

The design of the Paste Special dialog box assumes that if you are willing to permit a user to link to an object, you are also willing to permit the user to embed that object. For this reason, if any of the OLEUIPASTE_LINKTYPE flags associated with the **OLEUIPASTEFLAG** enumeration are set, then the OLEUIPASTE_PASTE flag must also be set in order for the data formats to appear in the Paste Special dialog box.

The text displayed in the Source field of the standard Paste Special dialog box, which is implemented in OLEDLG32.DLL, is the null-terminated string whose offset in bytes is specified in the *dwSrcofCopy* field of the **OBJECTDESCRIPTOR** structure for the object to be pasted. If an OBJECTDESCRIPTOR structure is not available for this object, the dialog box displays whatever text may be associated with **CF_LINKSOURCEDESCRIPTOR**. If neither structure is available, the dialog box looks for CF_FILENAME. If CF_FILENAME is not found, the dialog box displays the string "Unknown Source".

To free an HMETAFILEPICT returned from the Insert Object or Paste Special dialog box, delete the attached metafile on the handle, as follows:

```
void FreeHmetafilepict(HMETAFILEPICT hmfp)
{
    if (hmfp != NULL)
        {
        LPMETAFILEPICT pmfp = GlobalLock(hmfp);

        DeleteMetaFile(pmfp->hMF);
        GlobalUnlock(hmfp);
        GlobalFree(hmfp);
        }
}  // FreeHmetafilepict
```

See Also

OLEUIPASTEFLAG, **DeleteMetaFile**, **GlobalUnlock**, **GlobalFree** in Win32

OleUIPromptUser Quick Info

Displays a dialog box with the specified template and returns the response (button identifier) from the user. This function is used to display OLE warning messages, for example, Class Not Registered.

```
int OleUIPromtUser(
  int nTemplate,          //Resource number of dialog box
  HWND hwndParent         //Handle to the parent window of the dialog box
);
```

Parameters

nTemplate

[in] Resource number of the dialog box to display. See Remarks.

hwndParent

[in] Handle to the parent window of the dialog box. Specifies zero or more optional arguments. These parameters are passed to **wsprintf** to format the message string.

Return Values

Returns the button identifier selected by the user (template dependent).

Standard Success/Error Definitions

OLEUI_FALSE

Unknown failure (unused).

OLEUI_SUCCESS

No error, same as OLEUI_OK.

OLEUI_OK

The user pressed the OK button.

OLEUI_CANCEL

The user pressed the Cancel button.

Standard Field Validation Errors

OLEUI_ERR_STANDARDMIN

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

OLEUI_ERR_STRUCTURENULL

The pointer to an OLEUIXXX structure passed into the function was NULL.

OLEUI_ERR_STRUCTUREINVALID

Insufficient permissions for read or write access to an OLEUIXXX structure.

OLEUI_ERR_CBSTRUCTINCORRECT

The *cbstruct* value is incorrect.

OLEUI_ERR_HWNDOWNERINVALID

The *hWndOwner* value is invalid.

OLEUI_ERR_LPSZCAPTIONINVALID

The *lpszCaption* value is invalid.

OLEUI_ERR_LPFNHOOKINVALID

The *lpfnHook* value is invalid.

OLEUI_ERR_HINSTANCEINVALID

The *hInstance* value is invalid.

OLEUI_ERR_LPSZTEMPLATEINVALID

The *lpszTemplate* value is invalid.

OLEUI_ERR_HRESOURCEINVALID

The *hResource* value is invalid.

Initialization Errors

OLEUI_ERR_FINDTEMPLATEFAILURE

Unable to find the dialog box template.

OLEUI_ERR_LOADTEMPLATEFAILURE

Unable to load the dialog box template.

OLEUI_ERR_DIALOGFAILURE

Dialog box initialization failed.

OLEUI_ERR_LOCALMEMALLOC

A call to **LocalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_GLOBALMEMALLOC

A call to **GlobalAlloc** or the standard *IMalloc* allocator failed.

OLEUI_ERR_LOADSTRING

Unable to **LoadString** localized resources from the library.

OLEUI_ERR_OLEMEMALLOC

A call to the standard *IMalloc* allocator failed.

Function Specific Errors

OLEUI_ERR_STANDARDMAX

Errors common to all dialog boxes lie in the range OLEUI_ERR_STANDARDMIN to OLEUI_ERR_STANDARDMAX. This value allows the application to test for standard messages in order to display error messages to the user.

Remarks

The following dialog box templates are defined in *Windows Interface Guidelines--A Guide for Designing Software*. The *nTemplate* parameter must be a currently defined resource, however, additional templates can be added to prompt.dlg.

```
IDD_LINKSOURCEUNAVAILABLE
IDD_CANNOTUPDATELINK
IDD_SERVERNOTREG
IDD_CANNOTRESPONDVERB
IDD_SERVERNOTFOUND
IDD_UPDATELINKS
```

See Also

wsprintf in Win32

OleUIUpdateLinks Quick Info

Updates all links in the link container and displays a dialog box that shows the progress of the updating process. The process is stopped if the user presses the Stop button or when all links are processed.

```
BOOL OleUIUpdateLink(
  LPOLEUILINKCONTAINER lpOleUILinkCntr,   //Pointer to the link container
  HWND hwndParent,                        //Dialog box's parent window
  LPTSTR lpszTitle,                       //pointer to the dialog box title
  int cLinks                              //Number of links
);
```

Parameters

lpOleUILinkCntr

[in] Pointer to the **IOleUILinkContainer** interface on the link container.

hwndParent

[in] Parent window of the dialog box.

lpszTitle

[in] Pointer to the title of the dialog box.

cLinks

[in] Total number of links.

Return Values

TRUE

The links were successfully updated.

FALSE

Unable to update the links.

See Also

IOleUILinkContainer::GetLinkUpdateOptions, **IOleUILinkContainer::UpdateLink**

OleUninitialize Quick Info

Closes the OLE library, freeing any resources that it maintains.

void OleUninitialize();

Remarks

Call this function on application shutdown, as the last OLE library call. **OleUninitialize** calls the **CoUninitialize** function internally to shut down the OLE Component Object(COM) Library.

The **OleInitialize** and **OleUninitialize** calls must be balanced – if there are multiple calls to the **OleInitialize** function, there must be the same number of calls to **OleUninitialize**: Only the **OleUninitialize** call corresponding to the **OleInitialize** call that actually initialized the library can close it.

See Also

OleInitialize, **CoUninitialize**

ProgIDFromCLSID Quick Info

Retrieves the ProgID for a given CLSID.

```
WINOLEAPI ProgIDFromCLSID(
  REFCLSID clsid,              //CLSID for which the ProgID is requested
  LPOLESTR * lplpszProgID      //Indirect pointer to the requested ProgID on return
);
```

Parameters

clsid

[in] Specifies the CLSID for which the ProgID is requested.

lplpszProgID

[out] Indirect pointer to the requested ProgID.

Return Values

S_OK

The ProgID was returned successfully.

REGDB_E_CLASSNOTREG

Class not registered in the registry.

REGDB_E_READREGDB

Error reading registry.

Remarks

Every OLE object class listed in the Insert Object dialog box must have a *programmatic identifier* (ProgID), a string that uniquely identifies a given class, stored in the registry. In addition to determining the eligibility for the Insert Object dialog box, the ProgID can be used as an identifier in a macro programming language to identify a class. Finally, the ProgID is also the class name used for an object of an OLE class that is placed in an OLE 1 container.

The **ProgIDFromCLSID** function uses entries in the registry to do the conversion. OLE application authors are responsible for ensuring that the registry is configured correctly in the application's setup program.

The ProgID string must be different than the class name of any OLE 1 application, including the OLE 1 version of the same application, if there is one. In addition, a ProgID string must not contain more than 39 characters, start with a digit, or, except for a single period, contain any punctuation (including underscores).

The ProgID must *never* be shown to the user in the user interface. If you need a short displayable string for an object, call **IOleObject::GetUserType**.

Call the **CLSIDFromProgID** function to find the CLSID associated with a given ProgID. CLSIDs can be freed with the task allocator (refer to the **CoGetMalloc** function).

See Also

CLSIDFromProgID

PropagateResult Quick Info

```
#define PropagateResult(hrPrevious, scBase) ((HRESULT) scBase)
```

PropagateResult is a NO-OP.

This macro is obsolete and should not be used.

PropVariantClear Quick Info

Frees all elements that can be freed in a given PROPVARIANT structure.

```
HRESULT PropVariantClear(

  PROPVARIANT* pvarg      //Pointer to a
                          PROPVARIANT structure
);
```

Parameters

pvarg

[in] Pointer to an initialized PROPVARIANT structure for which any deallocatable elements are to be freed. On return, all zeroes are written to the PROPVARIANT.

Return Values

S_OK

The VT types are recognized and all items that can be freed have been freed.

STG_E_INVALID_PARAMETER

The variant has an unknown VT type.

Remarks

At any level of indirection, a NULL pointer is ignored. For example, in a VT_CF PROPVARIANT, the *pvarg*−*pclipdata*−*pClipData* could be NULL. In this case, the *pvarg*−*pclipdata*−*pClipData* pointer would be ignored, but the *pvarg*−*pclipdata* pointer would be freed.

On return, this function writes zeroes to the specified PROPVARIANT, so the VT-type is VT_EMPTY.

Passing NULL as the *pvarg* parameter produces a return code of S_OK.

See Also

FreePropVariantArray

PropVariantCopy Quick Info

Copies the contents of a PROPVARIANT structure to another.

```
HRESULT PropVariantCopy(
  PROPVARIANT * pDest      //Pointer to uninitialized PROPVARIANT that is filled
                           on return
  PROPVARIANT *pvarg       //PROPVARIANT to be copied
);
```

Parameters

pDest

[in, out] Pointer to an uninitialized **PROPVARIANT** structure that receives the copy.

pvarg

[in] Pointer to the PROPVARIANT to be copied.

Return Values

S_OK

The copy was successfully completed.

STG_E_INVALID_PARAMETER

The variant has an unknown type.

Remarks

Copies a PROPVARIANT by value so the original *pvarg* and new *pDest* may be freed independently with calls to **PropVariantClear**. For non-simple PROPVARIANT types such as VT_STREAM, VT_STORAGE, etc, which require a subobject, the copy is made by reference. The pointer is copied and AddRef is called on it. It is illegal to pass NULL for either *pDest* or *pvarg*.

See Also

PROPVARIANT, **PropVariantClear**

ReadClassStg Quick Info

Reads the CLSID previously written to a storage object with the **WriteClassStg**.

```
WINOLEAPI ReadClassStg(

  IStorage * pStg,      //Pointer to the storage object containing the CLSID
  CLSID * pclsid        //Pointer to return the CLSID
);
```

Parameters

pStg

[in] Pointer to the **IStorage** interface on the storage object containing the CLSID to be retrieved.

pclsid

[out] Pointer to where the CLSID is written. May return CLSID_NULL.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The CLSID was returned successfully.

This function also returns any of the error values returned by the **IStorage::Stat** method.

Remarks

This function is simply a helper function that calls the **IStorage::Stat** method and retrieves the CLSID previously written to the storage object with a call to **WriteClassStg** from the **STATSTG** structure.

See Also

OleLoad, **WriteClassStg**, **IStorage::Stat**, STATSTG structure.

ReadClassStm Quick Info

Reads the CLSID previously written to a stream object with the **WriteClassStm** method.

```
WINOLEAPI ReadClassStm(
  IStream * pStm,      //Pointer to the stream holding the CLSID
  CLSID * pclsid       //Pointer to where the CLSID is to be written
);
```

Parameters

pStm

[in] Pointer to the **IStream** interface on the stream object containing the CLSID to be read. This CLSID must have been previously written to the stream object using **WriteClassStm**.

pclsid

[out] Pointer to where the CLSID is to be written.

Return Values

S_OK

The CLSID was successfully retrieved.

STG_E_READFAULT

End of file was reached.

This function also returns any of the error values returned by the **IStream::Read** method.

Remarks

Most applications do not call the **ReadClassStm** method directly. OLE calls it before making a call to an object's **IPersistStream::Load** implementation.

See Also

WriteClassStm, **ReadClassStg**, **WriteClassStg**

ReadFmtUserTypeStg Quick Info

Returns the clipboard format and user type previously saved with the **WriteFmtUserTypeStg** function.

```
WINOLEAPI ReadFmtUserTypeStg(

  IStorage * pStg,              //Pointer to storage object holding the values
  CLIPFORMAT * pcf,             //Pointer to return the clipboard format
  LPWSTR * lplpszUserType       //Indirect pointer to return the user type string
);
```

Parameters

pStg

[in] Pointer to the **IStorage** interface on the storage object from which the information is to be read.

pcf

[out] Pointer to where the clipboard format is to be written on return. It can be NULL, indicating the format is of no interest to the caller.

lplpszUserType

[out] Indirect pointer to where the user type string is to be returned. It can be NULL, indicating that the user type is of no interest to the caller. This method allocates memory for the string. The caller is responsible for freeing the memory with **CoTaskMemFree**.

Return Values

This function supports the standard return values E_FAIL, E_INVALIDARG, and E_OUTOFMEMORY, as well as the following:

S_OK

The requested information was read successfully.

This function also returns any of the error values returned by the **IStream::Read** method.

Remarks

This function returns the clipboard format and the user type string from the specified storage object. The **WriteClassStg** function must have been called before calling the **ReadFmtUserTypeStg** function.

See Also

CoTaskMemFree, **WriteFmtUserTypeStg**

RegisterDragDrop Quick Info

Registers the specified window as one that can be the target of an OLE drag-and-drop operation and specifies the **IDropTarget** instance to use for drop operations.

```
WINOLEAPI RegisterDragDrop(
  HWND hwnd,                  //Handle to a window that can accept drops
  IDropTarget * pDropTarget   //Pointer to object that is to be target of drop
);
```

Parameters

hwnd

[in] Handle to a window that can be a target for an OLE drag-and-drop operation.

pDropTarget

[in] Pointer to the **IDropTarget** interface on the object that is to be the target of a drag-and-drop operation in a specified window. This interface is used to communicate OLE drag-and-drop information for that window.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

The application was registered successfully.

DRAGDROP_E_INVALIDHWND

Invalid handle returned in the *hwnd* parameter.

DRAGDROP_E_ALREADYREGISTERED

The specified window has already been registered as a drop target.

Remarks

If your application can accept dropped objects during OLE drag-and-drop operations, you must call the **RegisterDragDrop** function. Do this whenever one of your application windows is available as a potential drop target, i.e., when the window appears unobscured on the screen.

The **RegisterDragDrop** function only registers one window at a time, so you must call it for each application window capable of accepting dropped objects.

As the mouse passes over unobscured portions of the target window during an OLE drag-and-drop operation, the **DoDragDrop** function calls the specified **IDropTarget::DragOver** method for the current window. When a drop operation actually occurs in a given window, the **DoDragDrop** function calls **IDropTarget::Drop**.

The **RegisterDragDrop** function also calls the **IUnknown::AddRef** method on the **IDropTarget** pointer.

See Also

RevokeDragDrop

ReleaseStgMedium Quick Info

Frees the specified storage medium.

```
void ReleaseStgMedium(
  STGMEDIUM * pmedium      //Pointer to storage medium to be freed
);
```

Parameter

pmedium

[in] Pointer to the storage medium that is to be freed.

Return Value

None.

Remarks

The **ReleaseStgMedium** function calls the appropriate method or function to release the specified storage medium. Use this function during data transfer operations where storage medium structures are parameters, such as **IDataObject::GetData** or **IDataObject::SetData**. In addition to identifying the type of the storage medium, this structure specifies the appropriate **IUnknown::Release** method for releasing the storage medium when it is no longer needed.

It is common to pass a STGMEDIUM from one body of code to another, such as in **IDataObject::GetData**, in which the one called can allocate a medium and return it to the caller. **ReleaseStgMedium** permits flexibility in whether the receiving body of code owns the medium, or whether the original provider of the medium still owns it, in which case the receiving code needs to inform the provider that it can free the medium.

When the original provider of the medium is responsible for freeing the medium, the provider calls **ReleaseStgMedium**, specifying the medium and the appropriate **IUnknown** pointer as the *punkForRelease* structure member. Depending on the type of storage medium being freed, one of the following actions is taken, followed by a call to the **Release** method on the specified **IUnknown** pointer:

Medium	ReleaseStgMedium Action
TYMED_HGLOBAL	None.
TYMED_GDI	None.
TYMED_ENHMF	None.
TYMED_MFPICT	None.
TYMED_FILE	Frees the file name string using standard memory management mechanisms.
TYMED_ISTREAM	Calls **IStream::Release**.
TYMED_ISTORAGE	Calls **IStorage::Release**.

The provider indicates that the receiver of the medium is responsible for freeing the medium by specifying NULL for the *punkForRelease* structure member. Then the receiver calls **ReleaseStgMedium**, which makes a call as described in the following table depending on the type of storage medium being freed:

Medium	ReleaseStgMedium Action
TYMED_HGLOBAL	Calls the Win32 **GlobalFree** function on the handle.
TYMED_GDI	Calls the Win32 **DeleteObject** function on the handle.
TYMED_ENHMF	Deletes the enhanced metafile.
TYMED_MFPICT	The *hMF* that it contains is deleted with the Win32 **DeleteMetaFile** function; then the handle itself is passed to **GlobalFree**.
TYMED_FILE	Frees the disk file by deleting it. Frees the file name string by using the standard memory management paradigm.
TYMED_ISTREAM	**Calls IStream::Release**.
TYMED_ISTORAGE	**Calls IStorage::Release**.

In either case, after the call to **ReleaseStgMedium**, the specified storage medium is invalid and can no longer be used.

See Also

STGMEDIUM structure

ResultFromScode Quick Info

```
#define ResultFromScode(sc) ((HRESULT) (sc))
```

Converts an **SCODE** into an **HRESULT**.

This macro is obsolete and should not be used.

RevokeDragDrop Quick Info

Revokes the registration of the specified application window as a potential target for OLE drag-and-drop operations.

```
WINOLEAPI RevokeDragDrop(

  HWND hwnd      //Handle to a window that can accept drops
);
```

Parameter

hwnd

[in] Handle to a window previously registered as a target for an OLE drag-and-drop operation.

Return Values

This function supports the standard return value E_OUTOFMEMORY, as well as the following:

S_OK

Registration as a target window was revoked successfully.

DRAGDROP_E_INVALIDHWND

Invalid handle returned in the *hwnd* parameter.

DRAGDROP_E_NOTREGISTERED

An attempt was made to revoke a drop target that has not been registered.

Remarks

When your application window is no longer available as a potential target for an OLE drag-and-drop operation, you must call **RevokeDragDrop**.

This function calls the **IUnknown::Release** method for your drop target interface.

See Also

RegisterDragDrop

SCODE_CODE Quick Info

```
#define SCODE_CODE(sc) ((sc) & 0xFFFF)
```

Extracts the code part of the **SCODE**.

SCODE_FACILITY Quick Info

```
#define SCODE_FACILITY(sc) (((sc) >> 16) & 0x1fff)
```

Extracts the facility from the **SCODE**.

SCODE_SEVERITY Quick Info

```
#define SCODE_SEVERITY(sc) (((sc) >> 31) & 0x1)
```

Extracts the severity field from the **SCODE**.

SetConvertStg

Quick Info

Sets the convert bit in a storage object to indicate that the object is to be converted to a new class when it is opened. The setting can be retrieved with a call to the **GetConvertStg** function.

```
WINOLEAPI SetConvertStg(
  IStorage * pStg,        //Points to storage object where the conversion bit is to be set
  BOOL fConvert           //Indicates whether an object is to be converted
);
```

Parameters

pStg

IStorage pointer to the storage object in which to set the conversion bit.

fConvert

If TRUE, sets the conversion bit for the object to indicate the object is to be converted when opened. If FALSE, clears the conversion bit.

Return Values

S_OK

Indicates the object's conversion bit was set successfully.

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

E_OUTOFMEMORY

Indicates the conversion bit was not set due to a lack of memory.

E_INVALIDARG

Indicates one or more arguments are invalid.

E_UNEXPECTED

Indicates an unexpected error occurred.

See the **IStorage::CreateStream**, **IStorage::OpenStream**, **IStream::Read**, and **IStream::Write** methods for possible storage and stream access errors.

Remarks

The **SetConvertStg** function determines the status of the convert bit in a contained object. It is called by both the container application and the server in the process of converting an object from one class to another. When a user specifies through a **Convert To** dialogue (which the container produces with a call to the **OleUIConvert** function) that an object is to be converted, the container must take the following steps:

1. Unload the object if it is currently loaded.
2. Call **WriteClassStg** to write the new CLSID to the object storage.
3. Call **WriteFmtUserTypeStg** to write the new user type name and the existing main format to the storage.
4. Call **SetConvertStg** with the *fConvert* parameter set to TRUE to indicate that the object has been tagged for conversion to a new class the next time it is loaded.
5. Just before the object is loaded, call **OleDoAutoConvert** to handle any needed object conversion, unless you call **OleLoad**, which calls it internally.

When an object is initialized from a storage object and the server is the destination of a Convert To operation, the object server should do the following:

1. Call the **GetConvertStg** function to retrieve the value of the conversion bit.
2. If the bit is set, the server reads the data out of the object according to the format associated with the new CLSID.
3. When the object is asked to save itself, the object should call WriteFmtUserType() using the normal native format and user type of the object.
4. The object should then call **SetConvertStg** with the *fConvert* parameter set to FALSE to reset the object's conversion bit.

See Also

GetConvertStg

StgCreateDocfile Quick Info

Creates a new compound file storage object using the OLE-provided compound file implementation for the **IStorage** interface.

```
WINOLEAPI StgCreateDocfile(
  const WCHAR * pwcsName,          //Points to pathname of compound file to create
  DWORD grfMode,                   //Specifies the access mode for opening the storage object
  DWORD reserved,                  //Reserved; must be zero
  IStorage ** ppstgOpen            //Points to location for returning the new storage object
);
```

Parameters

pwcsName

[in] Points to the pathname of the compound file to create. It is passed uninterpreted to the file system. This can be a relative name or NULL. If NULL, a temporary compound file is allocated with a unique name.

grfMode

[in] Specifies the access mode to use when opening the new storage object. For more information, see the **STGM** enumeration. If the caller specifies transacted mode together with STGM_CREATE or STGM_CONVERT, the overwrite or conversion takes place at the time the storage object is opened and therefore is not revertible.

reserved

[in] Reserved for future use; must be zero.

ppstgOpen

[out] Points to the location of the **IStorage** pointer to the new storage object.

Return Values

S_OK

Indicates the compound file was successfully created.

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

STG_E_FILEALREADYEXISTS

Indicates the compound file already exists and *grfMode* is set to STGM_FAILIFTHERE.

STG_S_CONVERTED

Indicates the specified file was successfully converted to Storage format.

STG_E_INSUFFICIENTMEMORY

Indicates the compound file was not created due to a lack of memory.

STG_E_INVALIDNAME

Indicates bad name in the *pwcsName* parameter.

STG_E_INVALIDPOINTER

Indicates bad pointer in the *pwcsName* parameter or the *ppStgOpen* parameter.

STG_E_INVALIDFLAG

Indicates bad flag combination in the *grfMode* pointer.

STG_E_TOOMANYOPENFILES

Indicates the compound file was not created due to a lack of file handles.

See also any file system errors for other error return values.

Remarks

The **StgCreateDocfile** function creates a new storage object using the OLE-provided, compound-file implementation for the **IStorage** interface. The name of the open compound file can be retrieved by calling the **IStorage::Stat** method.

StgCreateDocfile creates the file if it does not exist. If it does exist, the use of the STGM_CREATE, STGM_CONVERT, and STGM_FAILIFTHERE flags in the *grfMode* parameter indicate how to proceed. See the **STGM** enumeration for more information on these values.

If the compound file is opened in transacted mode (the *grfMode* parameter specifies STGM_TRANSACTED) and a file with this name already exists, the existing file is not altered until all outstanding changes are committed. If the calling process lacks write access to the existing file (because of access control in the file system), the *grfMode* parameter can only specify STGM_READ and *not* STGM_WRITE or STGM_READWRITE. The resulting new open compound file can still be written to, but a commit operation will fail (in transacted mode, write permissions are enforced at commit time).

Specifying STGM_SIMPLE provides a much faster implementation of a compound file object in a limited, but frequently-used case. This can be used by applications that require a compound file implementation with multiple streams and no storages. The simple mode does not support all of the methods on **IStorage**. For more information, refer to the **STGM** enumeration.

If the *grfMode* parameter specifies STGM_TRANSACTED and no file yet exists with the name specified by the *pwcsName* parameter, the file is created immediately. In an access-controlled file system, the caller must have write permissions in the file system directory in which the compound file is created. If STGM_TRANSACTED is not specified, and STGM_CREATE is specified, an existing file with the same name is destroyed before creating the new file.

StgCreateDocfile can be used to create a temporary compound file by passing a NULL value for the *pwcsName* parameter. However, these files are temporary only in the sense that they have a system-provided unique name - likely one that is meaningless to the user. The caller is responsible for deleting the temporary file when finished with it, unless STGM_DELETEONRELEASE was specified for the *grfMode* parameter.

See Also

StgCreateDocFileOnILockBytes

StgCreateDocfileOnILockBytes Quick Info

Creates and opens a new compound file storage object on top of a byte array object provided by the caller. The storage object supports the OLE-provided, compound-file implementation for the **IStorage** interface.

```
WINOLEAPI StgCreateDocfileOnILockBytes(
  ILockBytes * plkbyt,          //Points to the ILockBytes interface on the byte array object
  DWORD grfMode,                //Specifies the access mode
  DWORD reserved,               //Reserved; must be zero
  IStorage ** ppstgOpen         //Points to location for returning the new storage object
);
```

Parameters

plkbyt

[in] Points to the **ILockBytes** interface on the underlying byte array object on which to create a compound file.

grfMode

[in] Specifies the access mode to use when opening the new compound file. For more information, see the **STGM** enumeration.

reserved

[in] Reserved for future use; must be zero.

ppstgOpen

[out] Points to the location of the **IStorage** pointer on the new storage object.

Return Values

S_OK

Indicates the compound file was successfully created.

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

STG_E_FILEALREADYEXISTS

Indicates the compound file already exists and the *grfMode* parameter is set to STGM_FAILIFTHERE.

STG_S_CONVERTED

Indicates the compound file was successfully converted. The original byte array object was successfully converted to **IStorage** format.

STG_E_INSUFFICIENTMEMORY

Indicates the storage object was not created due to a lack of memory.

STG_E_INVALIDPOINTER

Indicates a bad pointer was in the *pLkbyt* parameter or the *ppStgOpen* parameter.

STG_E_INVALIDFLAG

Indicates a bad flag combination was in the *grfMode* parameter.

STG_E_TOOMANYOPENFILES

Indicates the storage object was not created due to a lack of file handles.

See also any file system errors for other error return values.

See also the **ILockBytes** interface for other error return values.

Remarks

The **StgCreateDocfileOnILockBytes** function creates a storage object on top of a byte array object using the OLE-provided, compound-file implementation of the **IStorage** interface. **StgCreateDocfileOnILockBytes** can be used to store a document in a relational database. The byte array (indicated by the *pLkbyt* parameter, which points to the **ILockBytes** interface on the object) is used for the underlying storage in place of a disk file.

Except for specifying a programmer-provided byte-array object, **StgCreateDocfileOnILockBytes** is similar to the **StgCreateDocfile** function. For more information, refer to **StgCreateDocfile**.

The newly created compound file is opened according to the access modes in the *grfMode* parameter. For conversion purposes, the file is always considered to already exist. As a result, it is not useful to use the STGM_FAILIFTHERE value, because it always causes an error to be returned. However, STGM_CREATE and STGM_CONVERT are both still useful.

The ability to build a compound file on top of a byte array object is provided to support having the data (underneath an **IStorage** and **IStream** tree structure) live in a non-persistent space. Given this capability, there is nothing preventing a document that *is* stored in a file from using this facility. For example, a container might do this to minimize the impact on its file format caused by adopting OLE. However, it is recommended that OLE documents adopt the **IStorage** interface for their own outer-level storage. This has the following advantages:

- The storage structure of the document is the same as its storage structure when it is an embedded object, reducing the number of cases the application needs to handle.
- One can write tools to access the OLE embeddings and links within the document without special knowledge of the document's file format. An example of such a tool is a copy utility that copies all the documents included in a container containing linked objects. A copy utility like this needs access to the contained links to determine the extent of files to be copied.
- The **IStorage** implementation addresses the problem of how to commit the changes to the file. An application using the **ILockBytes** interface must handle these issues itself.
- Future file systems will likely implement the **IStorage** and **IStream** interfaces as their native abstractions, rather than layer on top of a byte array as is done in compound files. Such a file system could be built so documents using the **IStorage** interface as their outer level containment structure would get an automatic efficiency gain by having the layering flattened when files are saved on the new file system.

See Also

StgCreateDocfile

StgGetIFillLockBytesOnFile

Opens a wrapper object on a temporary file.

```
WINOLEAPI StgGetIFillLockBytes(

    OLECHAR *pwcsName          // Pointer to file specified by filename
    IFillLockBytes **ppflb     // Pointer to new byte array wrapper object
  );
```

Parameters

pwcsName

[in] Pointer to the name of the file to be instantiated.

pflb

[out] Indirect pointer to new byte array wrapper object.

Remarks

The moniker that manages the downloading of the file specified in *pwcsName* calls this function in the course of creating the asynchronous storage necessary to manage the asynchronous downloading of data. The moniker first creates a temporary file, then calls this function to create the wrapper object on that file. Finally, the moniker calls **StgOpenAsyncDocfileOnIFillLockBytes** to open the root storage of the compound file that is to be downloaded into the temporary file.

See Also

IFillLockBytes, **ILockBytes**, **StgOpenAsyncDocfileOnIFillLockBytes**

StgGetIFillLockBytesOnILockBytes

Creates a new wrapper object on a byte array object provided by the caller.

```
WINOLEAPI StgOpenAsyncDocFileOnIFillLockBytes(

    ILockBytes *pilb                // Pointer to existing byte array object
    IFillLockBytes **ppflb          // Pointer to new byte array wrapper object
  );
```

Parameters

pilb

[in] Pointer to an existing byte array object.

pflb

[out] Indirect pointer to new byte array wrapper object..

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as the following:

Remarks

The **StgGetIFillLockBytesOnILockBytes** function makes it possible to create an asynchronous storage wrapper object on a custom byte array object. For example, if you wanted to implement asynchronous storage on a database for which you have already created a byte array object, you would call this function to create the wrapper object for the byte array. To do so, the function instantiates a new wrapper object and then initializes it by passing it a pointer to the existing byte array object.

See Also

IFillLockBytes, **ILockBytes**

StgIsStorageFile Quick Info

Indicates whether a particular disk file contains a storage object.

```
WINOLEAPI StgIsStorageFile(
  const WCHAR * pwcsName          //Points to a pathname of the file to check
);
```

Parameter

pwcsName

[in] Points to the name of the disk file to be examined. The *pwcsName* parameter is passed uninterpreted to the underlying file system.

Return Values

S_OK

Indicates the file contains a storage object.

S_FALSE

Indicates the file does not contain a storage object.

STG_E_INVALIDFILENAME

Indicates a bad filename was passed in the *pwcsName* parameter.

STG_E_FILENOTFOUND

Indicates the file was not found.

See also any file system errors for other error return values.

Remarks

At the beginning of the disk file underlying a storage object is a signature distinguishing a storage object from other file formats. The **StgIsStorageFile** function is useful to applications whose documents use a disk file format that might or might not use storage objects.

If a root compound file has been created in transacted mode but not yet committed, this method will still return S_OK.

See Also

StgIsStorageILockBytes

StgIsStorageILockBytes Quick Info

Indicates whether the specified byte array contains a storage object.

```
WINOLEAPI StgIsStorageILockBytes(
  ILockBytes * plkbyt          //ILockBytes pointer to the byte array to be examined
);
```

Parameter

plkbyt

ILockBytes pointer to the byte array to be examined.

Return Values

S_OK

Indicates the specified byte array contains a storage object.

S_FALSE

Indicates the specified byte array does not contain a storage object.

File system errors.

ILockBytes interface error return values.

Remarks

At the beginning of the byte array underlying a storage object is a signature distinguishing a storage object (supporting the **IStorage** interface) from other file formats. The **StgIsStorageILockBytes** function is useful to applications whose documents use a byte array (a bye array object supports the **ILockBytes** interface) that might or might not use storage objects.

See Also

StgIsStorageFile, **ILockBytes**

StgOpenAsyncDocfileOnIFillLockBytes

Opens an existing root asynchronous storage object on a byte array wrapper object provided by the caller.

WINOLEAPI StgOpenAsyncDocFileOnIFillLockBytes(

IFillLockBytes **pflb*	// Pointer to byte array wrapper object
DWORD *grfmode*	// Storage access mode
DWORD *asyncFlags*	// Asynchronous storage flags
IStorage ***ppstgOpen*	// Indirect pointer to asynchronous storage

);

Parameters

pflb

[in] **IFillLockBytes** pointer to the byte array wrapper object that contains the storage object to be opened.

grfmode

[in] Specifies the access mode to use to open the storage object. The most common access mode, taken from the **STGM** enumeration, is STGM_READ.

asyncFlags

[in] Indicates whether a connection point on a storage will be inherited by its substorages and streams. ASYNC_MODE_COMPATIBILITY indicates that the connection point is inherited; ASYNC_MODE_DEFAULT indicates that the connection point is not inherited.

ppstgOpen

[out] Pointer to **IStorage** pointer on root asynchronous storage object.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as the following:

E_PENDING

Data is currently unavailable.

E_NOINTERFACE

A pointers was not returned to the requested interface.

STG_E_INSUFFICIENTMEMORY

There is insufficient memory to complete this operation.

Remarks

The root storage of the asynchronous storage object is opened according to the access mode in the *grfMode* parameter. A pointer to the **IStorage** interface on the opened storage object is supplied through the *ppstgOpen* parameter.

The byte array wrapper object must have been previously instantiated through a call to the **StgGetIFillLockBytesOnFile** function.

StgOpenAsyncDocFileOnIFillLockBytes does not support priority access mode or exclusions. Otherwise, it works in much the same way as the **StgOpenStorageOnILockBytes** function.

The returned storage object has a connection point for **IProgressNotify**.

See Also

IFillLockBytes, **ILockBytes**, **StgGetIFillLockBytesOnFile**, **StgOpenStorageOnILockBytes**.

StgOpenLayoutDocfile

Opens a compound file on an **ILockBytes** implementation that that is capable of monitoring sector information.

```
WINOLEAPI StgOpenLayoutDocfile(
    OLECHAR *pwcsName          // Pointer to name of compound file to be opened
    DWORD grfMode              // Access mode for the new storage object.
    DWORD reserved             // Reserved for future use.
    IStorage **ppstgOpen       // Indirect pointer to the new root storage object.
);
```

Parameters

pwcsName

[in] Pointer to the name of the compound file to be opened.

grfMode

[in] Access mode to use when opening the newly created storage object. Values are taken from the **STGM** enumeration. Note that priority mode and exclusions are not supported. The most common access mode is likely to be STGM_DIRECT | STGM_READ | STGM_SHARE_EXCLUSIVE.

ppstgOpen

[out] Indirect pointer to the **IStorage** interface on the root object of the newly created root storage object.

Return Values

This function supports the standard return values E_OUTOFMEMORY, E_UNEXPECTED, E_INVALIDARG, and E_FAIL, as well as the following:

STG_E_INVALIDPARAMETER

One of the parameters is invalid

STG_E_INVALIDNAME

The name passed to this function is not a valid filename

STG_E_INSUFFICIENTMEMORY

There is insufficient memory to complete this operation.

This function can also return any of the error values returned by the **StgOpenStorageOnILockBytes** function.

Remarks

The compound file implementation created by this function exposes the **ILayoutStorage** interface on its root storage. Applications use this interface to express the optimal layout of their compound files for the purpose of more rapidly downloading and rendering data over a slow link. **StgOpenLayoutDocfile** returns a pointer to the **IStorage** interface on the root storage of the newly created compound file. Using this pointer, applications call **QueryInterface** to obtain a pointer to **ILayoutStorage**.

See Also

ILockBytes, **IStorage**, **STGM**

StgOpenStorage Quick Info

Opens an existing root storage object in the file system. You can use this function to open compound files, but you cannot use it to open directories, files, or summary catalogs. Nested storage objects can only be opened using their parent's **IStorage::OpenStorage** method.

```
WINOLEAPI StgOpenStorage(
  const WCHAR * pwcsName,          //Points to the pathname of the file containing storage object
  IStorage * pstgPriority,         //Points to a previous opening of a root storage object
  DWORD grfMode,                   //Specifies the access mode for the object
  SNB snbExclude,                  //Points to an SNB structure specifying elements to be excluded
  DWORD reserved,                  //Reserved; must be zero
  IStorage ** ppstgOpen            //Indirect IStorage pointer to the storage object on return
);
```

Parameters

pwcsName

[in] Points to the pathname of the storage object to open. This parameter is ignored if the *pStgPriority* parameter is not NULL.

pstgPriority

[in] Most often NULL. If not NULL, this parameter is used instead of the *pwcsName* parameter to specify the pointer to the **IStorage** interface on the storage object to open. It points to a previous opening of a root storage object, most often one that was opened in priority mode.

After the **StgOpenStorage** function returns, the storage object specified in the *pStgPriority* parameter on function entry is invalid, and can no longer be used. Use the one specified in the *ppStgOpen* parameter instead.

grfMode

[in] Specifies the access mode to use to open the storage object.

snbExclude

[in] If not NULL, this parameter points to a block of elements in this storage that are to be excluded as the storage object is opened. This exclusion occurs independent of whether a snapshot copy happens on the open. May be NULL.

reserved

[in] Indicates reserved for future use; must be zero.

ppstgOpen

[out] Points to the location of the **IStorage** pointer on the opened storage.

Return Values

S_OK

Indicates the storage object was successfully opened.

STG_E_FILENOTFOUND

Indicates the specified file does not exist.

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

STG_E_FILEALREADYEXISTS

Indicates the file exists but is not a storage object.

STG_E_TOOMANYOPENFILES

Indicates the storage object was not opened because there are too many open files.

STG_E_INSUFFICIENTMEMORY

Indicates the storage object was not opened due to a lack of memory.

STG_E_INVALIDNAME

Indicates bad name in the *pwcsName* parameter.

STG_E_INVALIDPOINTER

Indicates bad pointer in one of the parameters: *snbExclude*, *pwcsName*, *pstgPriority*, or *ppStgOpen*.

STG_E_INVALIDFLAG

Indicates bad flag combination in the *grfMode* parameter.

STG_E_INVALIDFUNCTION

Indicates STGM_DELETEONRELEASE specified in the *grfMode* parameter.

STG_E_OLDFORMAT

Indicates the storage object being opened was created by the Beta 1 storage provider. This format is no longer supported.

STG_E_OLDDLL

Indicates the DLL being used to open this storage object is a version prior to the one used to create it.

STG_E_PATHNOTFOUND

Specified pathname does not exist.

Remarks

The **StgOpenStorage** function opens the specified root storage object according to the access mode in the *grfMode* parameter, and, if successful, supplies an **IStorage** pointer to the opened storage object in the *ppstgOpen* parameter.

Note Opening a storage object in read and/or write mode without denying writer permission to others (the *grfMode* parameter specifies STGM_SHARE_DENY_WRITE) can be a time-consuming operation since the **StgOpenStorage** call must make a snapshot of the entire storage object.

Applications will often try to open storage objects with the following access permissions:

```
STGM_READ_WRITE | STGM_SHARE_DENY_WRITE
    // transacted vs. direct mode omitted for exposition
```

If the application succeeds, it will never need to do a snapshot copy. If it fails, the application can revert to using the permissions and make a snapshot copy:

```
STGM_READ_WRITE
    // transacted vs. direct mode omitted for exposition
```

In this case, the application should prompt the user before doing a time-consuming copy. Alternatively, if the document sharing semantics implied by the access modes are appropriate, the application could try to open the storage as follows:

```
STGM_READ | STGM_SHARE_DENY_WRITE
    // transacted vs. direct mode omitted for exposition
```

In this case, if the application succeeds, a snapshot copy will not have been made (because STGM_SHARE_DENY_WRITE was specified, denying others write access).

To reduce the expense of making a snapshot copy, applications can open storage objects in priority mode (*grfMode* specifies STGM_PRIORITY).

The *snbExclude* parameter specifies a set of element names in this storage object that are to be emptied as the storage object is opened: streams are set to a length of zero; storage objects have all their elements removed. By excluding certain streams, the expense of making a snapshot copy can be significantly reduced. Almost always, this approach will be used after first opening the storage object in priority mode, then completely reading the now-excluded elements into memory. This earlier priority mode opening of the storage object should be passed through the *pstgPriority* parameter to remove the exclusion implied by priority mode. The calling application is responsible for rewriting the contents of excluded items before committing. Thus, this technique is most likely only useful to applications whose documents do not require constant access to their storage objects while they are active.

See Also

IStorage, **StgCreateDocfile**

StgOpenStorageOnILockBytes Quick Info

Opens an existing storage object that does not reside in a disk file, but instead has an underlying byte array provided by the caller.

```
WINOLEAPI StgOpenStorageOnILockBytes(
  ILockBytes * plkbyt,          //Points to the ILockBytes interface on the underlying byte array
  IStorage * pStgPriority,      //Points to a previous opening of a root storage object
  DWORD grfMode,                //Specifies the access mode for the object
  SNB snbExclude,               //Points to an SNB structure specifying elements to be excluded
  DWORD reserved,               //Reserved, must be zero
  IStorage ** ppstgOpen         //Points to location for returning the storage object
);
```

Parameters

plkbyt

[in] **ILockBytes** pointer to the underlying byte array object that contains the storage object to be opened.

pStgPriority

[in] Most often NULL. If not NULL, this parameter is used instead of the *plkbyt* parameter to specify the storage object to open. In this case, it points to the **IStorage** interface on a previously opened root storage object, most often one that was opened in priority mode.

After the **StgOpenStorageOnILockBytes** function returns, the storage object specified in the *pStgPriority* parameter on function entry is invalid, and can no longer be used; use the one specified in the *ppStgOpen* parameter instead.

grfMode

[in] Specifies the access mode to use to open the storage object.

snbExclude

[in] May be NULL. If not NULL, this parameter points to a block of elements in this storage that are to be excluded as the storage object is opened. This exclusion occurs independent of whether a snapshot copy happens on the open. .

reserved

[in] Indicates reserved for future use; must be zero.

ppstgOpen

[out] Points to the location of an **IStorage** pointer to the opened storage on successful return.

Return Values

S_OK

The storage object was successfully opened.

STG_E_FILENOTFOUND

The specified byte array does not exist.

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

STG_E_FILEALREADYEXISTS

The byte array exists but is not a storage object.

STG_E_TOOMANYOPENFILES

The storage object was not opened because there are too many open files.

STG_E_INSUFFICIENTMEMORY

The storage object was not opened due to a lack of memory.

STG_E_INVALIDNAME

Either *pwcsName* or *snbExclude* contains an invalid name.

STG_E_INVALIDPOINTER

Either *snbExclude*, *pwcsName*, *pstgPriority*, or *ppStgOpen contains an invalid pointer*.

STG_E_INVALIDFLAG

The *grfMode* parameter contains a bad flag combination.

STG_E_INVALIDFUNCTION

The access mode STGM_DELETEONRELEASE was specified in the *grfMode* parameter.

STG_E_OLDDLL

The DLL being used to open this storage object is a version prior to the one used to create it.

STG_E_OLDFORMAT

The storage object being opened was created by the Beta 1 storage provider. This format is no longer supported.

File system error return values.

ILockBytes interface error return values.

Remarks

StgOpenStorageOnILockBytes opens the specified root storage object. The storage object is opened according to the access mode in the *grfMode* parameter; a pointer to the IStorage interface on the opened storage object is supplied through the *ppstgOpen* parameter.

The storage object must have been previously created by the **StgCreateDocfileOnILockBytes** function.

Except for specifying a programmer-provided byte-array object, **StgOpenStorageOnILockBytes** is similar to the **StgOpenStorage** function. For more information, refer to **StgOpenStorage**.

See Also

StgOpenStorage, **StgCreateDocfileOnILockBytes**

StgSetTimes Quick Info

Sets the creation, access, and modification times of the indicated file, if supported by the underlying file system.

```
WINOLEAPI StgSetTimes(
  WCHAR const * lpszName,      //Points to the name of the file to be changed
  FILETIME const * pctime,     //Points to the new value for the creation time
  FILETIME const * patime,     //Points to the new value for the access time
  FILETIME const * pmtime      //Points to the new value for the modification time
);
```

Parameters

lpszName

[in] Points to the name of the file to be changed.

pctime

[in] Points to the new value for the creation time.

patime

[in] Points to the new value for the access time.

pmtime

[in] Points to the new value for the modification time.

Return Values

S_OK

Indicates time values successfully set.

STG_E_FILENOTFOUND

Indicates element does not exist.

STG_E_INVALIDNAME

Indicates bad name passed in the *lpszName* parameter, or a file system error.

STG_E_ACCESSDENIED

Access denied because the caller has insufficient permission, or another caller has the file open and locked.

STG_E_LOCKVIOLATION

Access denied because another caller has the file open and locked.

File system error return values.

Remarks

The **StgSetTimes** function sets the time values for the specified file. Each of the time value parameters can be NULL, indicating no modification should occur.

It is possible that one or more of these time values are not supported by the underlying file system. This function sets the times that can be set and ignores the rest.

StringFromCLSID Quick Info

Converts a CLSID into a string of printable characters. Different CLSIDs always convert to different strings.

```
WINOLEAPI StringFromCLSID(
  REFCLSID rclsid,      //CLSID to be converted
  LPOLESTR * ppsz       //Indirect pointer to the resulting string on return
);
```

Parameters

rclsid

[in] CLSID to be converted.

ppsz

[out] Pointer to the resulting string.

Return Values

This function supports the standard return value E_OUTOFMEMORY; as well as the following:

S_OK

Indicates the CLSID was successfully converted and returned.

Remarks

The **StringFromCLSID** function calls the **StringFromGuid2** function to convert a globally unique identifier (GUID) into a string of printable characters.

See Also

CLSIDFromString, **StringFromGuid2**

StringFromGUID2 Quick Info

Converts a globally unique identifier (GUID) into a string of printable characters.

```
StringFromGUID2(

  REFGUID rguid,      //Interface identifier to be converted
  LPOLESTR lpsz,      //Pointer to the resulting string on return
  int cbMax           //Maximum size the returned string is expected to be
);
```

Parameters

rguid

[in] Interface identifier to be converted.

lpsz

[out] Pointer to the resulting string on return.

cbMax

[in] Maximum size the returned string is expected to be.

Return Values

0 (zero)

Buffer is too small for returned string.

Non-zero value

The number of characters in the returned string, including the null terminator.

Remarks

The string that the *lpsz* parameter receives has a format like the following sample:

```
[c200e360-38c5-11ce-ae62-08002b2b79ef]
```

where the successive fields break the GUID into the form DWORD-WORD-WORD-WORD-WORD.DWORD covering the 128-bit GUID. The string includes enclosing braces, which are an OLE convention.

See Also

StringFromCLSID

StringFromIID Quick Info

Converts an interface identifier into a string of printable characters.

```
WINOLEAPI StringFromIID(
  REFIID rclsid          //Interface identifier to be converted
  LPOLESTR * lplpsz      //Indirect pointer to the resulting string
);
```

Parameters

rclsid

[in] Interface identifier to be converted.

lplpsz

[out] Indirect pointer to the resulting string on return.

Return Values

This function supports the standard return value E_OUTOFMEMORY; as well as the following:

S_OK

The character string was successfully returned.

Remarks

The string returned by the function is freed in the standard way, using the task allocator (refer to the **CoGetMalloc**function).

See Also

IIDFromString, **CoGetMalloc**

SUCCEEDED Quick Info

```
#define SUCCEEDED(Status) ((HRESULT)(Status) >= 0)
```

Provides a generic test for success on any status value. Non-negative numbers indicate success.

WriteClassStg Quick Info

Stores the specified CLSID in a storage object.

```
WINOLEAPI WriteClassStg(
  IStorage * pStg,          //Points to the IStorage interface on the storage object
  REFCLSID rclsid           //Specifies the CLSID to be stored in the storage object
);
```

Parameters

pStg

[in] **IStorage** pointer to the storage object that will get a new CLSID.

rclsid

[in] Points to the CLSID to be stored with the object.

Return Values

S_OK

Indicates the CLSID was successfully written to the file.

STG_E_MEDIUMFULL

Indicates the CLSID could not be written due to lack of memory.

IStorage::SetClass method error return values.

Remarks

The **WriteClassStg** function writes a CLSID to the specified storage object so it can be read by the **ReadClassStg** function. Container applications typically call this function before calling the **IPersistStorage::Save** method.

See Also

OleSave, **ReadClassStg**

WriteClassStm Quick Info

Stores the specified CLSID in the stream.

```
WINOLEAPI WriteClassStm(
  IStream * pStm,          //Points to the IStream interface on the stream object
  REFCLSID rclsid          //Specifies the CLSID to be stored in the stream object
);
```

Parameters

pStm

[in] **IStream** pointer to the stream into which the CLSID is to be written.

rclsid

[in] Specifies the CLSID to write to the stream.

Return Values

S_OK

Indicates the CLSID was successfully written.

STG_E_MEDIUMFULL

The CLSID could not be written because there is no space left on device.

IStorage::SetClass method error return values.

Remarks

The **WriteClassStm** function writes a CLSID to the specified stream object so it can be read by the **ReadClassStm** function. Most applications do not call **WriteClassStm** directly. OLE calls it before making a call to an object's **IPersistStream::Save** method.

See Also

ReadClassStm, **WriteClassStg**, **ReadClassStg**

WriteFmtUserTypeStg Quick Info

Writes a clipboard format and user type to the storage object.

```
WINOLEAPI WriteFmtUserTypeStg(
  IStorage * pStg,          //Points to the IStorage interface on the storage object
  CLIPFORMAT cf,            //Specifies the clipboard format
  LPWSTR * lpszUserType     //Points to the current user type
);
```

Parameters

pStg

[in] **IStorage** pointer to the storage object where the information is to be written.

cf

[in] Specifies the clipboard format that describes the structure of the native area of the storage object. The format tag includes the policy for the names of streams and substorages within this storage object and the rules for interpreting data within those streams.

lpszUserType

[in] Points to the object's current user type. It cannot be NULL. This is the type returned by the **IOleObject::GetUserType** method. If this function is transported to a remote machine where the object class does not exist, this persistently stored user type can be shown to the user in dialog boxes.

Return Values

S_OK

Indicates the information was written successfully.

STG_E_MEDIUMFULL

Indicates information could not be written due to lack of space on the storage medium.

IStream::Write method error return values.

Remarks

The **WriteFmtUserTypeStg** function must be called in an object's implementation of the **IPersistStorage::Save** method. It must also be called by document-level objects that use structured storage for their persistent representation in their save sequence.

To read the information saved, applications call the **ReadFmtUserTypeStg** function.

See Also

IPersistStorage::Save, **ReadFmtUserTypeStg**

BIND_OPTS Quick Info

Contains parameters used during a moniker-binding operation. The **BIND_OPTS2** structure may be used in its place. A **BIND_OPTS** structure is stored in a bind context; the same bind context is used by each component of a composite moniker during binding, allowing the same parameters to be passed to all components of a composite moniker. See **IBindCtx** for more information about bind contexts.

If you're a moniker client (that is, you use a moniker to acquire an interface pointer to an object), you typically do not need to specify values for the fields of this structure. The **CreateBindCtx** function creates a bind context with the bind options set to default values that are suitable for most situations; the **BindMoniker** function does the same thing when creating a bind context for use in binding a moniker. If you want to modify the values of these bind options, you can do so by passing a **BIND_OPTS** structure to the **IBindCtx::SetBindOptions** method. Moniker implementers can pass a **BIND_OPTS** structure to the **IBindCtx::GetBindOptions** method to retrieve the values of these bind options.

The **BIND_OPTS** structure is defined in OBJIDL.IDL.

```
typedef struct tagBIND_OPTS
{
    DWORD cbStruct;
    DWORD grfFlags;
    DWORD grfMode;
    DWORD dwTickCountDeadline;
} BIND_OPTS, *LPBIND_OPTS;
```

Members

cbStruct

Size of this structure in bytes (that is, the size of the **BIND_OPTS** structure).

grfFlags

Flags that control aspects of moniker binding operations. This value is any combination of the bit flags in the **BINDFLAGS** enumeration. New values may be defined in the future, so moniker implementations should ignore any bits in this field that they do not understand. The **CreateBindCtx** function initializes this field to zero.

grfMode

Flags that should be used when opening the file that contains the object identified by the moniker. The values are taken from the **STGM** enumeration. The binding operation uses these flags in the call to **IPersistFile::Load** when loading the file. If the object is already running, these flags are ignored by the binding operation. The **CreateBindCtx** function initializes this field to STGM_READWRITE.

dwTickCountDeadline

Clock time (in milliseconds, as returned by the **GetTickCount** function) by which the caller would like the binding operation to be completed. This member lets the caller limit the execution time of an operation when speed is of primary importance. A value of zero indicates that there is no deadline. Callers most often use this capability when calling the **IMoniker::GetTimeOfLastChange** method, though it can be usefully applied to other operations as well. The **CreateBindCtx** function initializes this field to zero.

Typical deadlines allow for a few hundred milliseconds of execution. This deadline is a recommendation, not a requirement; however, operations that exceed their deadline by a large amount may cause delays for the end user. Each moniker implementation should try to complete its operation by the deadline, or fail with the error MK_E_EXCEEDEDDEADLINE.

If a binding operation exceeds its deadline because one or more objects that it needs are not running,

the moniker implementation should register the objects responsible in the bind context using the **IBindCtx::RegisterObjectParam**. The objects should be registered under the parameter names "ExceededDeadline", "ExceededDeadline1", "ExceededDeadline2", and so on. If the caller later finds the object in the Running Object Table, the caller can retry the binding operation.

The **GetTickCount** function indicates the number of milliseconds since system startup, and wraps back to zero after 2^31 milliseconds. Consequently, callers should be careful not to inadvertently pass a zero value (which indicates no deadline), and moniker implementations should be aware of clock wrapping problems (see the **GetTickCount** function for more information).

See Also

BIND_OPTS2, **BIND_FLAGS**, **CreateBindCtx**, **IBindCtx::SetBindOptions**

BIND_OPTS2

Contains parameters used during a moniker-binding operation. A **BIND_OPTS2** structure is stored in a bind context; the same bind context is used by each component of a composite moniker during binding, allowing the same parameters to be passed to all components of a composite moniker. See **IBindCtx** for more information about bind contexts. BIND_OPTS2 replaces the previously defined **BIND_OPTS** structure, including the previously defined members, and adding four new members.

Moniker clients (those using a moniker to acquire an interface pointer to an object) typically do not need to specify values for the fields of this structure. The **CreateBindCtx** function creates a bind context with the bind options set to default values that are suitable for most situations. The **BindMoniker** function does the same thing when creating a bind context for use in binding a moniker. If you want to modify the values of these bind options, you can do so by passing a **BIND_OPTS2** structure to the **IBindCtx::SetBindOptions** method. Moniker implementers can pass a **BIND_OPTS2** structure to the **IBindCtx::GetBindOptions** method to retrieve the values of these bind options.

The **BIND_OPTS2** structure is defined in OBJIDL.IDL

```
typedef struct tagBIND_OPTS2 {
    DWORD           cbStruct;       //  sizeof(BIND_OPTS2)
    DWORD           grfFlags;
    DWORD           grfMode;
    DWORD           dwTickCountDeadline;
    DWORD           dwTrackFlags;
    DWORD           dwClassContext;
    LCID            locale;
    COSERVERINFO *  pServerInfo;
} BIND_OPTS2, * LPBIND_OPTS2;
```

Members

cbStruct

Size of this structure in bytes (that is, the size of the **BIND_OPTS2** structure).

grfFlags

Flags that control aspects of moniker binding operations. This value is any combination of the bit flags in the **BINDFLAGS** enumeration. New values may be defined in the future, so moniker implementations should ignore any bits in this field that they do not understand. The **CreateBindCtx** function initializes this field to zero.

grfMode

Flags that should be used when opening the file that contains the object identified by the moniker. The values are taken from the **STGM** enumeration. The binding operation uses these flags in the call to **IPersistFile::Load** when loading the file. If the object is already running, these flags are ignored by the binding operation. The **CreateBindCtx** function initializes this field to STGM_READWRITE.

dwTickCountDeadline

Clock time (in milliseconds, as returned by the **GetTickCount** function) by which the caller would like the binding operation to be completed. This member lets the caller limit the execution time of an operation when speed is of primary importance. A value of zero indicates that there is no deadline. Callers most often use this capability when calling the **IMoniker::GetTimeOfLastChange** method, though it can be usefully applied to other operations as well. The **CreateBindCtx** function initializes this field to zero.

Typical deadlines allow for a few hundred milliseconds of execution. This deadline is a recommendation, not a requirement; however, operations that exceed their deadline by a large amount may cause delays for the end user. Each moniker implementation should try to complete its operation by the deadline, or fail with the error MK_E_EXCEEDEDDEADLINE.

If a binding operation exceeds its deadline because one or more objects that it needs are not running, the moniker implementation should register the objects responsible in the bind context using the **IBindCtx::RegisterObjectParam**. The objects should be registered under the parameter names "ExceededDeadline", "ExceededDeadline1", "ExceededDeadline2", and so on. If the caller later finds the object in the Running Object Table, the caller can retry the binding operation.

The **GetTickCount** function indicates the number of milliseconds since system startup, and wraps back to zero after 2^31 milliseconds. Consequently, callers should be careful not to inadvertently pass a zero value (which indicates no deadline), and moniker implementations should be aware of clock wrapping problems (see the **GetTickCount** function for more information).

dwTrackFlags

A moniker can use this value during link tracking. If the original persisted data that the moniker is referencing has been moved, the moniker can attempt to reestablish the link by searching for the original data though some adequate mechanism. *dwTrackFlags* provides additional information on how the link should be resolved. See the documentation of the *fFlags* parameter in **IShellLink::Resolve** in the Win32 SDK for more details.

COM's file moniker implementation uses the shell link mechanism to reestablish links and passes these flags to **IShellLink::Resolve**.

dwClassContext

The class context, taken from the CLSCTX enumeration, that is to be used for instantiating the object. Monikers typically pass this value to the *dwClsContext* parameter of **CoCreateInstance**.

locale

The LCID value indicating the client's preference for the locale to be used by the object to which they are binding. A moniker passes this value to **IClassActivator::GetClassObject**.

pServerInfo

Points to a **COSERVERINFO** structure. This member allows clients calling **IMoniker::BindToObject** to specify server information. Clients may pass a BIND_OPTS2 structure to the **IBindCtx::SetBindOptions** method. If a server name is specified in the COSERVERINFO struct, the moniker bind will be forwarded to the specified machine. **SetBindOptions** only copies the struct members of **BIND_OPTS2**, not the **COSERVERINFO** structure and the pointers it contains. Callers may not free any of these pointers until the bind context is released. COM's new class moniker does not currently honor the *pServerInfo* flag.

See Also

BIND_OPTS, **BIND_FLAGS**, **CreateBindCtx**, **IBindCtx::SetBindOptions**

CADWORD Quick Info

The **CADWORD** structure is a counted array of DWORDs. It is used, for example, in the **IPerPropertyBrowsing::GetPredefinedStrings** method. The values returned in this counted array can be passed into the **IPerPropertyBrowsing::GetPredefinedValue** method to obtain the value corresponding to one of the predefined strings for a property.

```
typedef struct tagCADWORD
{
    ULONG      cElems;
    DWORD FAR* pElems;
} CADWORD;
```

Members

cElems

Size of the array pointed to by *pElems*.

pElems

Pointer to an array of DWORD values, each of which can be passed to the **IPerPropertyBrowsing::GetPredefinedValue** method to obtain the corresponding value for one of the property's predefined strings. This array is allocated by the callee using **CoTaskMemAlloc** and is freed by the caller using **CoTaskMemFree**.

See Also

CALPOLESTR, **IPerPropertyBrowsing::GetPredefinedStrings**, **IPerPropertyBrowsing::GetPredefinedValue**

CALPOLESTR Quick Info

The **CALPOLESTR** structure is a counted array of LPOLESTR types, that is, a counted array of pointers to strings. It is used, for example, in the **IPerPropertyBrowsing::GetPredefinedStrings** method to specify the predefined strings that a property can accept.

```
typedef struct tagCALPOLESTR
{
    ULONG         cElems;
    LPOLESTR FAR* pElems;
} CALPOLESTR;
```

Members

cElems

Size of the array pointed to by *pElems*.

pElems

Pointer to an array of LPOLESTR values, each of which corresponds to an allowable value that a particular property can accept. The caller can use these string values in user interface elements, such as drop-down list boxes. This array, as well as the strings in the array, are allocated by the callee using **CoTaskMemAlloc** and is freed by the caller using **CoTaskMemFree**.

See Also

IPerPropertyBrowsing::GetPredefinedStrings

CAUUID Quick Info

The **CAUUID** structure is a Counted Array of UUID or GUID types. It is used, for example, in the **ISpecifyPropertyPages::GetPages** method to receive an array of CLSIDs for the property pages that the object wants displayed.

```
typedef struct tagCAUUID
{
    ULONG     cElems;
    GUID FAR* pElems;
} CAUUID;
```

Members

cElems

Size of the array pointed to by *pElems*.

pElems

Pointer to an array of UUID values, each of which specifies a CLSID of a particular property page. This array is allocated by the callee using **CoTaskMemAlloc** and is freed by the caller using **CoTaskMemFree**.

See Also

ISpecifyPropertyPages::GetPages

COAUTHINFO

Determines the authentication settings used while making a remote activation request from the client scm to the server.

```
typedef struct _COAUTHINFO
{
    DWORD               dwAuthnSvc;
    DWORD               dwAuthzSvc;
    [string] WCHAR *    pwszServerPrincName;
    DWORD               dwAuthnLevel;
    DWORD               dwImpersonationLevel;
    AUTH_IDENTITY  *    pAuthIdentityData;
    DWORD               dwCapabilities;
} COAUTHINFO;
```

Members

dwAuthnSvc

[in] A single DWORD value from the list of **RPC_C_AUTHN_*xxx*** constants indicating the *authentication* service to use. It may be RPC_C_AUTHN_NONE if no authentication is required.

dwAuthzSvc

[in] A single DWORD value from the list of **RPC_C_AUTHZ_*xxx*** constants indicating the *authorization* service to use. If you are using the NT authentication service, use RPC_C_AUTHZ_NONE.

pwszServerPrincName

Pointer to a WCHAR string that indicates the server principal name to use with the authentication service. If you are using RPC_C_AUTHN_WINNT, the principal name must be NULL.

dwAuthnLevel

[in] A single DWORD value from the list of **RPC_C_AUTHN_LEVEL_*xxx*** constants indicating the authentication level to use.

dwImpersonationLevel

[in] A single DWORD value from the list of **RPC_C_IMP_LEVEL_*xxx*** constants indicating the impersonation level to use. Currently, only RPC_C_IMP_LEVEL_IMPERSONATE and RPC_C_IMP_LEVEL_IDENTIFY are supported.

pAuthIdentityData

Pointer to an **AUTH_IDENTITY** structure that establishes the identity of the client. It is authentication-service specific. It is in the form of Windows NT's SEC_WINNT_AUTH_IDENTITY, as follows:

```
typedef struct _AUTH_IDENTITY
{
  [size_is(UserLength+1)] USHORT *        User;
  ULONG                   UserLength;
  [size_is(DomainLength+1)] USHORT *      Domain;
  ULONG                   DomainLength;
  [size_is(PasswordLength+1)] USHORT *    Password;
  ULONG                   PasswordLength;
  ULONG                   Flags;
} AUTH_IDENTITY;
```

dwCapabilities

[in] A DWORD defining flags to establish indicating the further capabilities of this proxy. Currently, no capability flags are defined.

Remarks

The values of the **COAUTHINFO** structure determine the authentication settings used while making a remote activation request from the client's scm to the server's scm. This structure is defined by default for NTLMSSP, and is described only for cases that need it to allow DCOM activations to work correctly with security providers other than NTLMSSP, or to specify additional security information used during remote activations for interoperability with alternate implementations of distributed COM. Currently, the impersonation level must be set to RPC_C_IMP_LEVEL_IMPERSONATE, or the result will be a failed activation when the server is running WindowsNT.

See Also

COSERVERINFO

CONNECTDATA Quick Info

The **CONNECTDATA** structure is the type enumerated through the **IEnumConnections::Next** method. Each structure describes a connection that exists to a given connection point.

```
typedef struct tagCONNECTDATA
{
    IUnknown*  pUnk;
    DWORD      dwCookie;
} CONNECTDATA;
```

Members

pUnk

Pointer to the **IUnknown** interface on a connected advisory sink. The caller must call **IUnknown::Release** using this pointer when the **CONNECTDATA** structure is no longer needed. The caller is responsible for calling **Release** for each **CONNECTDATA** structure enumerated through **IEnumConnections::Next**.

dwCookie

Connection where this value is the same token that is returned originally from calls to **IConnectionPoint::Advise**. This token can be used to disconnect the sink pointed to by a *pUnk* by passing *dwCookie* to **IConnectionPoint::Unadvise**.

See Also

IConnectionPoint, **IEnumConnections**

CONTROLINFO Quick Info

The **CONTROLINFO** structure contains parameters that describe a control's keyboard mnemonics and keyboard behavior. The structure is filled during the **IOleControl::GetControlInfo** method.

```
typedef struct tagCONTROLINFO
    {
    ULONG  cb;
    HACCEL hAccel;
    USHORT cAccel;
    DWORD  dwFlags;
    } CONTROLINFO;
```

Members

cb

Size of the **CONTROLINFO** structure.

hAccel

Handle to an array of Windows **ACCEL** structures, each structure describing a keyboard mnemonic. The array is allocated with the **GlobalAlloc** function. The control always maintains the memory for this array; the caller of **IOleControl::GetControlInfo** should not attempt to free the memory.

cAccel

Number of mnemonics described in the *hAccel* field. This value can be zero to indicate no mnemonics.

dwFlags

Flags that indicate the keyboard behavior of the control. The possible values are:

CTRLINFO_EATS_RETURN

When the control has the focus, it will process the Return key.

CTRLINFO_EATS_ESCAPE

When the control has the focus, it will process the Escape key.

When the control has the focus, the dialog box containing the control cannot use the Return or Escape keys as mnemonics for the default and cancel buttons.

See Also

IOleControl::GetControlInfo

COSERVERINFO Quick Info

Identifies a remote machine resource to the new or enhanced activation functions. The structure is defined as follows in the Wtypes.h header file:

```
typedef struct  _COSERVERINFO
    {
    DWORD dwReserved1;
    LPWSTR pwszName;
    COAUTHINFO  *pAuthInfo;
    DWORD dwReserved2;
    }   COSERVERINFO;
```

Members

dwReserved1

Reserved for future use. Must be 0.

pszName

Pointer to the name of the machine to be used.

pAuthInfo

When using NTLMSSP, this value must be set to zero. A non-zero value, which is a pointer to a **COAUTHINFO** structure, would only be used when a security package other than NTLMSSP is being used.

dwReserved2

Reserved for future use. Must be 0.

Remarks

The COSERVERINFO structure is used primarily to identify a remote system in object creation functions. Machine resources are named using the naming scheme of the network transport. By default, all UNC ("\\ server" or "server") and DNS names ("server.com", "www.foo.com", or "135.5.33.19") names are allowed.

If you are using the NTLMSSP security package, the default case, the *pAuthinfo* parameter should be set to zero. If you are a vendor supporting another security package, refer to **COAUTHINFO**. The mechanism described there is intended to allow DCOM activations to work correctly with security providers other than NTLMSSP, or to specify additional security information used during remote activations for interoperability with alternate implementations of DCOM. If *pAuthInfo* is set, those values will be used to specify the authentication settings for the remote call. These settings will be passed to **RpcBindingSetAuthInfoEx**.

If the *pAuthInfo* field is not specified, any values in the **AppID** section of the registry will be used to override the following default authentication settings:

dwAuthnSvc	RPC_C_AUTHN_WINNT
dwAuthzSvc	RPC_C_AUTHZ_NONE
pszServerPrincName	NULL
dwAuthnLevel	RPC_C_AUTHN_LEVEL_CONNECT
dwImpersonationLevel	RPC_C_IMP_LEVEL_IMPERSONATE
pvAuthIdentityData	NULL
dwCapabilities	RPC_C_QOS_CAPABILITIES_DEFAULT

See Also

CLSCTX, **CoGetClassObject**, **CoGetInstanceFromFile**, **CoGetInstanceFromIStorage**, **CoCreateInstanceEx**, **Locating a Remote Object**

DVASPECTINFO Quick Info

The **DVASPECTINFO** structure is used in the **IViewObject::Draw** method to optimize rendering of an inactive object by making more efficient use of the GDI. The *pvAspect* parameter in **IViewObject::Draw** points to this structure. It is defined as follows:

```
typedef struct STRUCT tagDVASPECTINFO
{
    UNIT  cb;
    DWORD  dwFlags;
} DVASPECTINFO;
```

Members

cb

Size of the structure in bytes. The size includes this member as well as the *dwFlags* member.

dwFlags

A value taken from the **DVASPECTINFOFLAG** enumeration.

See Also

DVASPECTINFOFLAG

DVEXTENTINFO Quick Info

The **DVEXTENTINFO** structure is used in **IViewObjectEx::GetNaturalExtent**.

```
typedef struct tagDVEXTENTINFO
{
    ULONG cb;
    DWORD dwExtentMode;
    SIZEL sizelProposed;
}DVEXTENTINFO;
```

Members

cb

Size of the structure in bytes. The size includes this member as well as the *dwExtentMode* and *sizelProposed* members.

dwExtentMode

Indicates whether the sizing mode is content or integral sizing. See the **DVEXTENTMODE** enumeration for these values.

sizelProposed

Specifies the proposed size in content sizing or the preferred size in integral sizing.

See Also

DVEXTENTMODE

DVTARGETDEVICE Quick Info

Use the **DVTARGETDEVICE** structure to specify information about the target device for which data is being composed. **DVTARGETDEVICE** contains enough information about a Windows target device so a handle to a device context (hDC) can be created using the Windows **CreateDC** function.

```
typedef struct tagDVTARGETDEVICE
{
    DWORD tdSize;
    WORD  tdDriverNameOffset;
    WORD  tdDeviceNameOffset;
    WORD  tdPortNameOffset;
    WORD  tdExtDevmodeOffset;
    BYTE  tdData[1];
}DVTARGETDEVICE;
```

Members

tdSize

Size, in bytes, of the **DVTARGETDEVICE** structure. The initial size is included so the structure can be copied more easily.

tdDriverNameOffset

Offset, in bytes, from the beginning of the structure to the device driver name, which is stored as a NULL-terminated string in the *tdData* buffer.

tdDeviceNameOffset

Offset, in bytes, from the beginning of the structure to the device name, which is stored as a NULL-terminated string in the *tdData* buffer. This value can be zero to indicate no device name.

tdPortNameOffset

Offset, in bytes, from the beginning of the structure to the port name, which is stored as a NULL-terminated string in the *tdData* buffer. This value can be zero to indicate no port name.

tdExtDevmodeOffset

Offset, in bytes, from the beginning of the structure to the **DEVMODE** structure retrieved by calling **ExtDeviceMode**.

tdData

Aray of bytes containing data for the target device. It is not necessary to include empty strings in *tdData* (for names where the offset value is zero).

Remarks

Some OLE 1 client applications incorrectly construct target devices by allocating too few bytes in the **DEVMODE** structure for the OLETARGETDEVICE. They typically only supply the number of bytes in the DEVMODE.*dmSize* member. The number of bytes to be allocated should be the sum of DEVMODE.*dmSize* + DEVMODE.*dmDriverExtra*. When a call is made to the **CreateDC** function with an incorrect target device, the printer driver tries to access the additional bytes and unpredictable results can occur. To protect against a crash and make the additional bytes available, OLE pads the size of OLE 2 target devices created from OLE 1 target devices.

See Also

FORMATETC, **IEnumFORMATETC**, **IViewObject**, **OleConvertOLESTREAMToIStorage**

FILETIME Quick Info

The **FILETIME** data structure is a 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601. It is the means by which Win32 determines the date and time. **FILETIME** is used by the **CoDosDateTimeToFileTime**, **CoFileTimeToDosDateTime**, and **CoFileTimeNow** functions. It is defined as follows:

```
typedef struct STRUCT tagFILETIME
{
    DWORD  dwLowDateTime;
    DWORD  dwHighDateTime;
} FILETIME;
```

Members

dwLowDateTime

The low 32 bits of the Win32 date/time value.

dwHighDateTime

The upper 32 bits of the Win32 date/time value.

Remarks

The **FILETIME** data structure is used in the time conversion functions between DOS and Win32.

See Also

CoDosDateTimeToFileTime, **CoFileTimeNow**, **CoFileTimeToDosDateTime**

FONTDESC Quick Info

The **FONTDESC** structure contains parameters used to create a font object through the **OleCreateFontIndirect** function.

```
typedef struct tagFONTDESC
    {
    UINT cbSizeOfStruct;
    LPOLESTR lpstrName;
    CY cySize;
    SHORT sWeight;
    SHORT sCharset;
    BOOL fItalic;
    BOOL fUnderline;
    BOOL fStrikethrough;
    } FONTDESC;
```

Members

cbSizeOfStruct

Size of the **FONTDESC** structure.

lpstrName

Pointer to the caller-owned string specifying the font name.

cySize

Initial point size of the font in CY units.

sWeight

Initial weight of the font. If the weight is below 550 (the average of FW_NORMAL, 400, and FW_BOLD, 700), then the Bold property is also initialized to FALSE. If the weight is above 550, the Bold property is set to TRUE.

sCharset

Initial character set of the font.

fItalic

Initial italic state of the font.

fUnderline

Initial underline state of the font.

fStrikethrough

Initial strikethrough state of the font.

See Also

OleCreateFontIndirect

FORMATETC Quick Info

The **FORMATETC** structure is a generalized Clipboard format. It is enhanced to encompass a target device, the aspect or view of the data, and a storage medium indicator. Where one might expect to find a Clipboard format, OLE uses a **FORMATETC** data structure instead. This structure is used as a parameter in OLE functions and methods that require data format information.

```
typedef struct tagFORMATETC
{
    CLIPFORMAT      cfFormat;
    DVTARGETDEVICE  *ptd;
    DWORD           dwAspect;
    LONG            lindex;
    DWORD           tymed;
}FORMATETC, *LPFORMATETC;
```

Members

cfFormat

Particular clipboard format of interest. There are three types of formats recognized by OLE:

- Standard interchange formats, such as CF_TEXT.
- Private application formats understood only by the application offering the format, or by other applications offering similar features.
- OLE formats, which are used to create linked or embedded objects.

ptd

Pointer to a **DVTARGETDEVICE** structure containing information about the target device for which the data is being composed. A NULL value is used whenever the specified data format is independent of the target device or when the caller doesn't care what device is used. In the latter case, if the data requires a target device, the object should pick an appropriate default device (often the display for visual components). Data obtained from an object with a NULL target device, such as most metafiles, is independent of the target device. The resulting data is usually the same as it would be if the user chose the Save As command from the File menu and selected an interchange format.

dwAspect

One of the **DVASPECT** enumeration constants that indicate how much detail should be contained in the rendering. A single clipboard format can support multiple aspects or views of the object. Most data and presentation transfer and caching methods pass aspect information. For example, a caller might request an object's iconic picture, using the metafile clipboard format to retrieve it. Note that only one **DVASPECT** value can be used in *dwAspect*. That is, *dwAspect* cannot be the result of a BOOLEAN OR operation on several **DVASPECT** values.

lindex

Part of the aspect when the data must be split across page boundaries. The most common value is -1, which identifies all of the data. For the aspects DVASPECT_THUMBNAIL and DVASPECT_ICON, *lindex* is ignored.

tymed

One of the **TYMED** enumeration constants which indicate the type of storage medium used to transfer the object's data. Data can be transferred using whatever medium makes sense for the object. For example, data can be passed using global memory, a disk file, or structured storage objects. For more information, see the **TYMED** enumeration.

Remarks

The **FORMATETC** structure is used by methods in the data transfer and presentation interfaces as a parameter specifying the data being transferred. For example, the **IDataObject::GetData** method uses the **FORMATETC** structure to indicate exactly what kind of data the caller is requesting.

See Also

DVASPECT, **IDataAdviseHolder**, **IDataObject**, **IEnumFORMATETC**, **IOleCache**, **OleCreate**, **OleCreateFromData**, **OleCreateLinkFromData**, **OleCreateStaticFromData**, **OleCreateLink**, **OleCreateLinkToFile**, **OleCreateFromFile**, **STATDATA**, **STGMEDIUM**, **TYMED**

INTERFACEINFO Quick Info

The **INTERFACEINFO** structure contains information about incoming calls. The structure is defined as follows:

```
typedef struct tagINTERFACEINFO
{
    LPUNKNOWN  pUnk;
    IID        iid;
    WORD       wMethod;
} INTERFACEINFO,  * LPINTERFACEINFO;
```

Members

pUnk

Pointer to the **IUnknown** interface on the object.

iid

Identifier of the requested interface

wMethod

Interface method.

See Also

IMessageFilter::HandleIncomingCall

LICINFO Quick Info

The **LICINFO** structure contains parameters that describe the licensing behavior of a class factory that supports licensing. The structure is filled during the **IClassFactory2::GetLicInfo** method.

```
typedef struct tagLICINFO
    {
    ULONG cbLicInfo;
    BOOL  fRuntimeKeyAvail;
    BOOL  fLicVerified;
    } LICINFO;
```

Members

cbLicInfo

Size of the **LICINFO** structure.

fRuntimeKeyAvail

Whether this class factory allows the creation of its objects on a unlicensed machine through the use of a license key. If TRUE, **IClassFactory2::RequestLicKey** can be called to obtain the key. If FALSE, objects can be created only on a fully licensed machine.

fLicVerified

Whether a full machine license exists so that calls to **IClassFactory::CreateInstance** and **IClassFactory2::RequestLicKey** will succeed. If TRUE, the full machine license exists. Thus, objects can be created freely. and a license key is available if *fRuntimeKeyAvail* is also TRUE. If FALSE, this class factory cannot create any instances of objects on this machine unless the proper license key is passed to **IClassFactory2::CreateInstanceLic**.

See Also

IClassFactory::CreateInstance, **IClassFactory2::CreateInstanceLic**, **IClassFactory2::GetLicInfo**, **IClassFactory2::RequestLicKey**

MULTI_QI Quick Info

To optimize network performance, most remote activation functions take an array of MULTI_QI structures rather than just a single IID as input and a single pointer to the requested interface on the object as output, as do local machine activation functions. This allows a set of pointers to interfaces to be returned from the same object in a single round-trip to the server. In network scenarios, requesting multiple interfaces at the time of object construction can save considerable time over using a number of calls to the **QueryInterface** method for unique interfaces, each of which would require a round-trip to the server.

```
typedef struct _MULTI_QI {
        const IID*    pIID;
        IUnknown *    pItf;
        HRESULT       hr;
    } MULTI_QI;
```

Members

pIID

[in] Pointer to an interface identifier.

pItf

[out] Pointer to the interface requested in *pIID*. Must be set to NULL on entry.

hr

[out] Return value of the **QueryInterface** call made to satisfy the request for the interface requested in *pIID*. Common return values are S_OK and E_NOINTERFACE. Must be set to zero on entry.

See Also

CoGetInstanceFromFile, **CoGetInstanceFromIStorage**, **CoCreateInstanceEx**

OBJECTDESCRIPTOR Quick Info

The **OBJECTDESCRIPTOR** structure is the data structure used for the CF_OBJECTDESRIPTOR and CF_LINKSRCDESCRIPTOR file formats. These formats provide user interface information during data transfer operations, for example, the Paste Special dialog box or target feedback information during drag-and-drop operations.

```
typedef struct tagOBJECTDESCRIPTOR
{
    ULONG  cbSize;
    CLSID  clsid;
    DWORD  dwDrawAspect;
    SIZEL  sizel;
    POINTL pointl;
    DWORD  dwStatus;
    DWORD  dwFullUserTypeName;
    DWORD  dwSrcOfCopy;
    /* variable sized string data may appear here */
} OBJECTDESCRIPTOR;
```

Members

cbSize

Size of structure in bytes.

clsid

CLSID of the object being transferred. The *clsid* is used to obtain the icon for the Display As Icon option in the Paste Special dialog box and is applicable only if the Embed Source or Embedded Object formats are offered. If neither is offered, the value of *clsid* should be CLSID_NULL. The *clsid* can be retrieved by the source by loading the object and calling the **IOleObject::GetUserClassID** method. Note that for link objects, this value is not the same as the value returned by the **IPersist::GetClassID** method.

dwDrawAspect

Display aspect of the object. Typically, this value is DVASPECT_CONTENT or DVASPECT_ICON. If the source application did not draw the object originally, the *dwDrawAspect* field contains a zero value (which is not the same as DVASPECT_CONTENT).

sizel

True extent of the object (without cropping or scaling) in HIMETRIC units. Setting this field is optional. The value can be (0,0) for applications that do not draw the object being transferred. This field is used primarily by targets of drag-and-drop operations, so they can give appropriate feedback to the user.

pointl

Offset in HIMETRIC units from the upper-left corner of the object where a drag-and-drop operation was initiated. This field is only meaningful for a drag-and-drop transfer operation since it corresponds to the point where the mouse was clicked to initiate the drag-and-drop operation. The value is (0,0) for other transfer situations, such as a Clipboard copy and paste.

dwStatus

Copy of the status flags for the object. These flags are defined by the **OLEMISC** enumeration. If an embedded object is being transferred, they are returned by calling the **IOleObject::GetMiscStatus** method.

dwFullUserTypeName

Offset for finding the full user type name of the object being transferred. It specifies the offset, in bytes, from the beginning of the **OBJECTDESCRIPTOR** data structure to the null-terminated string that specifies the full user type name of the object being transferred. The value is zero if the string is not present. This string is used by the destination of a data transfer to create labels in the Paste Special dialog box. The destination application must be able to handle the cases when this string is omitted.

dwSrcOfCopy

Offset, in bytes, from the beginning of the data structure to the null-terminated string that specifies the source of the transfer. The *dwSrcOfCopy* field is typically implemented as the display name of the temporary moniker that identifies the data source. The value for *dwSrcOfCopy* is displayed in the Source line of the Paste Special dialog box. A zero value indicates that the string is not present. If *dwSrcOfCopy* is zero, the string "Unknown Source" is displayed in the Paste Special dialog box.

See Also

IDataObject, **FORMATETC**

OCPFIPARAMS Quick Info

The **OCPFIPARAMS** structure contains parameters used to invoke a property sheet dialog box through the **OleCreatePropertyFrameIndirect** function.

```
typedef struct tagOCPFIPARAMS
{
    ULONG          cbStructSize;
    HWND           hWndOwner;
    int            x;
    int            y;
    LPCOLESTR      lpszCaption;
    ULONG          cObjects;
    LPUNKNOWN FAR* lplpUnk;
    ULONG          cPages;
    CLSID FAR*     lpPages;
    LCID           lcid;
    DISPID         dispIDInitialProperty;
} OCPFIPARAMS;
```

Members

cbStructsize

Size of the **OCPFIPARAMS** structure.

hWndOwner

Parent window of the resulting property sheet dialog box.

x

Horizontal position for the dialog box relative to *hWndOwner*.

y

Vertical position for the dialog box relative to *hWndOwner*.

lpszCaption

Pointer to the string used for the caption of the dialog.

cObjects

Number of object pointers passed in *lplpUnk*.

lplpUnk

Array of **IUnknown** pointers on the objects for which this property sheet is being invoked. The number of elements inthe aray is specified by *cObject*. These pointers are passed to each property page through **IPropertyPage::SetObjects**.

cPages

Number of property pages specified in *lpPages*.

lpPages

Pointer to an array of size *cPages* containing the **CLSID**s of each property page to display in the property sheet.

lcid

Locale identifier for the property sheet. This value will be returned through **IPropertyPageSite::GetLocaleID**.

dispIDInitialProperty

Property that is highlighted when the dialog box is made visible.

See Also

IPropertyPage::SetObjects, **IPropertyPageSite::GetLocaleID**, **OleCreatePropertyFrameIndirect**

OLEINPLACEFRAMEINFO Quick Info

The **OLEINPLACEFRAMEINFO** structure contains information about the accelerators supported by a container during an in-place session. The structure is used in the **IOleInPlaceSite::GetWindowContext** method and the **OleTranslateAccelerator** function.

```
typedef struct tagOIFI
{
    UINT    cb;
    BOOL    fMDIApp;
    HWND    hwndFrame;
    HACCEL haccel;
    UINT    cAccelEntries;
} OLEINPLACEFRAMEINFO, *LPOLEINPLACEFRAMEINFO;
```

Members

cb

Size in bytes of this structure. The object server must specify sizeof(**OLEINPLACEFRAMEINFO**) in the structure it passes to **IOleInPlaceSite::GetWindowContext**. The container can then use this size to determine the structure's version.

fMDIApp

Whether the container is an MDI application.

hwndFrame

Handle to the container's top-level frame window.

haccel

Handle to the accelerator table that the container wants to use during an in-place editing session.

cAccelEntries

Number of accelerators in *haccel*.

Remarks

When an object is being in-place activated, its server calls the container's **IOleInPlaceSite::GetWindowContext** method, which fills in an **OLEINPLACEFRAMEINFO** structure. During an in-place session, the message loop of an EXE server passes a pointer to the **OLEINPLACEFRAMEINFO** structure to **OleTranslateAccelerator**. OLE uses the information in this structure to determine whether a message maps to one of the container's accelerators.

See Also

IOleInPlaceSite::GetWindowContext, **OleTranslateAccelerator**

OLEMENUGROUPWIDTHS Quick Info

The **OLEMENUGROUPWIDTHS** structure is the mechanism for building a shared menu. It indicates the number of menu items in each of the six menu groups of a menu shared between a container and an object server during an in-place editing session.

The structure is defined in the **IOleInPlaceFrame** interface (*inplcf.idl*). It is used in the **IOleInPlaceFrame::InsertMenus** and **ICDStandardForm::SetMenu** methods, and the **OleCreateMenuDescriptor** function.

```
typedef struct tagOleMenuGroupWidths
{
    LONG width[6];
} OLEMENUGROUPWIDTHS, * LPOLEMENUGROUPWIDTHS;
```

Member

width

An array whose elements contain the number of menu items in each of the six menu groups of a shared in-place editing menu. Each menu group can have any number of menu items. The container uses elements 0, 2, and 4 to indicate the number of menu items in its File, View, and Window menu groups. The object server uses elements 1, 3, and 5 to indicate the number of menu items in its Edit, Object, and Help menu groups.

Remarks

A container application and an object server use this structure to build a shared menu. The object server initializes to zeros the array elements in an **OLEMENUGROUPWIDTHS** structure and passes a pointer to it along with a menu handle to the container in a call to **IOleInPlaceFrame::InsertMenus**. The container adds its menu items to the menu, and fills in the structure with the number of items in each of its groups (indexes 0, 2, and 4). The server then uses the group width values returned by the container to insert its menu items in the appropriate position in the menu. The server fills in the structure with the number of items in each of its groups (indexes 1, 3, and 5), and then passes the structure to OLE in a call to the **OleCreateMenuDescriptor** function. This enables OLE to intercept the container's menu messages and redirect the messages generated by the server's menus.

See Also

IOleInPlaceFrame::InsertMenus, **OleCreateMenuDescriptor**

OLEUIBUSY Quick Info

The **OLEUIBUSY** structure contains information that the OLE User Interface Library uses to initialize the Busy dialog box, and space for the library to return information when the dialog box is dismissed.

```
typedef struct tagOLEUIBUSY
{
// These IN fields are standard across all OLEUI dialog box functions.
    DWORD          cbStruct;
    DWORD          dwFlags;
    HWND           hWndOwner;
    LPCSTR         lpszCaption;
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    HINSTANCE      hInstance;
    LPCSTR         lpszTemplate;
    HRSRC          hResource;

// Specifics for OLEUIBUSY.
    HTASK  hTask;
    HWND FAR *  lphWndDialog;
} OLEUICHANGEICON, *POLEUICHANGEICON, FAR *LPOLEUICHANGEICON;
```

Members

cbStruct

Size of the structure in bytes. This field must be filled on input.

dwFlags

On input, specifies the initialization and creation flags. On exit, it specifies the user's choices. It may be a combination of the following flags:

BZ_DISABLECANCELBUTTON

Input only: This flag disables the Cancel button.

BZ_DISABLESWITCHTOBUTTON

Input only: This flag disables the Switch To... button.

BZ_DISABLERETRYBUTTON

Input only: This flag disables the Retry button.

BZ_NOTRESPONDINGDIALOG

Input only: This flag generates a Not Responding dialog box instead of a Busy dialog box. The text is slightly different, and the Cancel button is disabled.

hWndOwner

Window that owns the dialog box. It should not be NULL.

lpszCaption

Pointer to a string to be used as the title of the dialog box. If NULL, then the library uses Busy.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function

must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook* member. The library passes a pointer to the **OLEUIBUSY** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the *lCustData* member.

hInstance

Instance that contains a dialog box template specified by the *lpTemplateName* member.

lpszTemplate

Pointer to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the library's Busy dialog box template.

hResource

Customized template handle.

hTask Input only:

Handle to the task that is blocking.

lphWndDialog

Pointer to the dialog box's HWND.

See Also

OleUIBusy

OLEUICHANGEICON Quick Info

The **OLEUICHANGEICON** structure contains information that the OLE User Interface Library uses to initialize the Change Icon dialog box, and it contains space for the library to return information when the dialog box is dismissed.

```
typedef struct tagOLEUICHANGEICON
{
// These IN fields are standard across all OLEUI dialog box functions.
    DWORD          cbStruct;
    DWORD          dwFlags;
    HWND           hWndOwner;
    LPCSTR         lpszCaption;
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    HINSTANCE      hInstance;
    LPCSTR         lpszTemplate;
    HRSRC          hResource;

 // Specifics for OLEUICHANGEICON.
    HGLOBAL  hMetaPict;
    CLSID    clsid;
    char     szIconExe[OLEUI_CCHPATHMAX];
    int      cchIconExe;
} OLEUICHANGEICON, *POLEUICHANGEICON, FAR *LPOLEUICHANGEICON;
```

Members

cbStruct

Size of the structure in bytes. This field must be filled on input.

dwFlags

On input, specifies the initialization and creation flags. On exit, it specifies the user's choices. It can be a combination of the following flags:

CIF_SHOWHELP

Dialog box will display a Help button.

CIF_SELECTCURRENT

On input, selects the Current radio button on initialization. On exit, specifies that the user selected Current.

CIF_SELECTDEFAULT

On input, selects the Default radio button on initialization. On exit, specifies that the user selected Default.

CIF_SELECTFROMFILE

On input, selects the From File radio button on initialization. On exit, specifies that the user selected From File.

CIF_USEICONEXE

Input only. Extracts the icon from the executable specified in the *szIconExe* member, instead of retrieving it from the class. This is useful for OLE embedding or linking to non-OLE files.

hWndOwner

Window that owns the dialog box. It should not be NULL.

lpszCaption

Pointer to a string to be used as the title of the dialog box. If NULL, then the library uses Change Icon.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook* member. The library passes a pointer to the **OLEUICHANGEICON** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the *lCustData* member.

hInstance

Instance that contains a dialog box template specified by the *lpTemplateName* member.

lpszTemplate

Pointer to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the library's Change Icon dialog box template.

hResource

Customized template handle.

hMetaPict

Current and final image. The source of the icon is embedded in the metafile itself.

clsid

Input only. The class to use to get the Default icon.

szIconExe

Input only. Pointer to the executable to extract the default icon from. This member is ignored unless CIF_USEICONEXE is included in the *dwFlags* parameter and an attempt to retrieve the class icon from the specified CLSID fails.

cchIconExe

Input only. The number of characters in *szIconExe*. This member is ignored unless CIF_USEICONEXE is included in the *dwFlags* member.

See Also

OleUIChangeIcon

OLEUICHANGESOURCE Quick Info

This structure is used to initialize the standard Change Source dialog box. It allows the user to modify the destination or source of a link. This may simply entail selecting a different file name for the link, or possibly changing the item reference within the file, for example, changing the destination range of cells within the spreadsheet that the link is to.

```
typedef struct tagOLEUICHANGESOURCEW
{
// These IN fields are standard across all OLEUI dialog box functions.
    DWORD          cbStruct;
    DWORDT         dwFlags;
    HWND           hWndOwner;
    LPCWSTR        lpszCaption;
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    HINSTANCE      hInstance;
    LPCWSTR        lpszTemplate;
    HRSRC          hResource;

// INTERNAL ONLY: do not modify these members
    OPENFILENAMEW*     lpOFN;
    DWORD              dwReserved1[4];

// Specifics for OLEUICHANGESOURCE.
    LPOLEUILINKCONTAINERW  lpOleUILinkContainer;
    DWORD                  dwLink;
    LPTSTR                 lpszDisplayName;
    ULONG                  nFileLength;
    LPTSTR                 lpszFrom;
    LPTSTR                 lpszTo;
} OLEUICHANGESOURCEW, *POLEUICHANGESOURCEW, FAR *LPOLEUICHANGESOURCEW;
```

Members

cbStruct

Size of the structure in bytes.

dwFlags

On input, this field specifies the initialization and creation flags. On exit, it specifies the user's choices. It may be a combination of the following flags:

CSF_SHOWHELP

Enables or shows the Help button.

CSF_VALIDSOURCE

Indicates that the link was validated.

CSF_ONLYGETSOURCE

Disables automatic validation of the link source when the user presses OK. If you specify this flag, you should validate the source when the dialog box returns OK.

hWndOwner

Window that owns the dialog box.

lpszCaption

Pointer to a string to be used as the title of the dialog box. If NULL, then the library uses Change Source.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook* member. The library passes a pointer to the **OLEUICHANGEICON** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the *lCustData* member.

hInstance

Instance that contains a dialog box template specified by the *lpszTemplate* member. This member is ignored if the *lpszTemplate* member is NULL or invalid.

lpszTemplate

Pointer to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the library's Convert dialog box template.

hResource

Resource handle for a custom dialog box. If this member is NULL, then the library uses the standard Convert dialog box template, or if it is valid, the template named by the *lpszTemplate* member.

lpOFN

Pointer to the **OPENFILENAME** structure, which contains information used by the operating system to initialize the system-defined Open or Save As dialog boxes.

dwReserved1[4]

Reserved for future use.

lpOleUILinkContainer

Pointer to the container's implementation of the **IOleUILinkContainer i**nterface, used to validate the link source. The Edit Links dialog box uses this to allow the container to manipulate its links.

dwLink

Container-defined unique 32-bit link identifier used to validate link sources. Used by *lpOleUILinkContainer*.

lpszDisplayName

Pointer to the complete source display name.

nFileLength

File moniker portion of *lpszDisplayName*.

lpszFrom

Pointer to the prefix of the source that was changed from.

lpszTo

Pointer to the prefix of the source to be changed to.

See Also

IOleUILinkContainer, OleUIChangeSource

OLEUICONVERT Quick Info

The **OLEUICONVERT** structure contains information that the OLE User Interface Library uses to initialize the Convert dialog box, and space for the library to return information when the dialog box is dismissed.

```
typedef struct tagOLEUICONVERT
{
// These IN fields are standard across all OLEUI dialog functions.
    DWORD          cbStruct
    DWORD          dwFlags
    HWND           hWndOwner
    LPCSTR         lpszCaption
    LPFNOLEUIHOOK  lpfnHook
    LPARAM         lCustData
    HINSTANCE      hInstance
    LPCSTR         lpszTemplate
    HRSRC          hResource

// Specifics for OLEUICONVERT.
    CLSID    clsid;
    CLSID    clsidConvertDefault;
    CLSID    clsidActivateDefault;
    CLSID    clsidNew;
    DWORD    dvAspect;
    WORD     wFormat;
    BOOL     fIsLinkedObject;
    HGLOBAL  hMetaPict;
    LPTSTR   lpszUserType;
    BOOL     fObjectsIconChanged;
    LPTSTR   lpszDefLabel
    UINT     cClsidExclude
    LPCLSID  lpClsidExclude
} OLEUICONVERT, *POLEUICONVERT, FAR *LPOLEUICONVERT;
```

Members

cbStruct

Size of the structure, in bytes. This field must be filled on input.

dwFlags

On input, this field specifies the initialization and creation flags. On exit, it specifies the user's choices. It may be a combination of the following flags:

CF_SHOWHELPBUTTON

Dialog box will display a Help button. This flag is set on input.

CF_SETCONVERTDEFAULT

Class whose CLSID is specified by *clsidConvertDefault* will be used as the default selection. This selection appears in the class listbox when the Convert To radio button is selected. This flag is set on input.

CF_SETACTIVATEDEFAULT

Class whose CLSID is specified by *clsidActivateDefault* will be used as the default selection. This

selection appears in the class listbox when the Activate As radio button is selected. This flag is set on input.

CF_SELECTCONVERTTO

On input, this flag specifies that Convert To will be initially selected (default behavior). This flag is set on output if Convert To was selected when the user dismissed the dialog box.

CF_SELECTACTIVATEAS

On input, this flag specifies that Activate As will be initially selected. This flag is set on output if Activate As was selected when the user dismissed the dialog box.

CF_DISABLEDISPLAYASICON

The Display As Icon button will be disabled on initialization.

CF_DISABLEACTIVATEAS

The Activate As radio button will be disabled on initialization.

CF_HIDECHANGEICON

The Change Icon button will be hidden in the Convert dialog box.

CF_CONVERTONLY

The Activate As radio button will be disabled in the Convert dialog box.

hWndOwner

Window that owns the dialog box. It should not be NULL.

lpszCaption

Pointer to a string to be used as the title of the dialog box. If NULL, then the library uses Convert.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook* member. The library passes a pointer to the **OLEUICONVERT** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the *lCustData* member.

hInstance

Instance that contains a dialog box template specified by the *lpszTemplate* member. This member is ignored if the *lpszTemplate* member is NULL or invalid.

lpszTemplate

Pointer to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the library's Convert dialog box template.

hResource

Resource handle for a custom dialog box. If this member is NULL, then the library uses the standard Convert dialog box template, or if it is valid, the template named by the *lpszTemplate* member.

clsid

The CLSID of the object to be converted or activated. This member is set on input.

clsidConvertDefault

The CLSID to use as the default class when Convert To is selected. This member is ignored if the *dwFlags* member does not include CF_SETCONVERTDEFAULT. This member is set on input.

clsidActivateDefault

The CLSID to use as the default class when Activate As is selected. This member is ignored if the *dwFlags* member does not include CF_SETACTIVATEDEFAULT. This member is set on input.

clsidNew

The CLSID of the selected class. This member is set on output.

dvAspect

Aspect of the object. This must be either DVASPECT_CONTENT or DVASPECT_ICON. If *dvAspect* is DVASPECT_ICON on input, then the Display As Icon box is checked and the object's icon is displayed. This member is set on input and output.

wFormat

Data format of the object to be converted or activated.

fIsLinkedObject

TRUE if the object is linked. This member is set on input.

hMetaPict

The METAFILEPICT containing the iconic aspect. This member is set on input and output.

lpszUserType

Pointer to the User Type name of the object to be converted or activated. If this value is NULL, then the dialog box will retrieve the User Type name from the registry. This string is freed on exit.

fObjectsIconChanged

TRUE if the object's icon changed. (that is, if **OleUIChangeIcon** was called and not canceled.). This member is set on output.

lpszDefLabel

Pointer to the default label to use for the icon. If NULL, the short user type name will be used. If the object is a link, the caller should pass the Display Name of the link source. This is freed on exit.

cClsidExclude

Number of CLSIDs in *lpClsidExclude*.

lpClsidExclude

Pointer to the list of CLSIDs to exclude from the list.

See Also

OleUIConvert, **OleUIChangeIcon**

OLEUIEDITLINKS Quick Info

The **OLEUIEDITLINKS** structure contains information that the OLE User Interface Library uses to initialize the Edit Links dialog box, and contains space for the library to return information when the dialog box is dismissed.

```
typedef struct tagOLEUIEDITLINKS
{
// These IN fields are standard across all OLEUI dialog box functions.
    DWORD          cbStruct;
    DWORD          dwFlags;
    HWND           hWndOwner;
    LPCSTR         lpszCaption;
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    HINSTANCE      hInstance;
    LPCSTR         lpszTemplate;
    HRSRC          hResource;

// Specifics for OLEUIEDITLINKS.
    LPOLEUILINKCONTAINER      lpOleUILinkContainer;
} OLEUIEDITLINKS, *POLEUIEDITLINKS, FAR *LPOLEUIEDITLINKS;
```

Members

cbStruct

Size of the structure in bytes. This field must be filled on input.

dwFlags

On input, *dwFlags* specifies the initialization and creation flags. It may be a combination of the following flags:

ELF_SHOWHELP

Specifies that the dialog box will display a Help button.

ELF_DISABLEUPDATENOW

Specifies that the Update Now button will be disabled on initialization.

ELF_DISABLEOPENSOURCE

Specifies that the Open Source button will be disabled on initialization.

ELF_DISABLECHANGESOURCE

Specifies that the Change Source button will be disabled on initialization.

ELF_DISABLECANCELLINK

Specifies that the Cancel Link button will be disabled on initialization.

hWndOwner

Window that owns the dialog box. It should not be NULL.

lpszCaption

Pointer to a string to be used as the title of the dialog box. If NULL, then the library uses Links.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook* member. The library passes a pointer to the **OLEUIEDITLINKS** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the *lCustData* member.

hInstance

Instance that contains a dialog box template specified by the *lpTemplateName* member.

lpszTemplate

Pointer to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the library's Edit Links dialog box template.

hResource

Customized template handle.

lpOleUILinkContainer

Pointer to the container's implementation of the **IOleUILinkContainer** Interface. The Edit Links dialog box uses this to allow the container to manipulate its links.

See Also

IOleUILinkContainer, **OleUIEditLinks**

OLEUIGNRLPROPS Quick Info

This structure is used to initialize the General tab of the Object Properties dialog box. A reference to it is passed in as part of the **OLEUIOBJECTPROPS** structure to the **OleUIObjectProperties** function. This tab shows the type and size of an OLE embedding and allows it the user to tunnel to the Convert dialog box. This tab also shows the link destination if the object is a link.

```
typedef struct tagOLEUIGNRLPROPS
{
// These IN fields are standard across all OLEUI property pages.
    DWORD          cbStruct;
    DWORD          dwFlags;
    DWORD          dwReserved1[2];
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    DWORD          dwReserved2[3];
    struct tagOLEUIOBJECTPROPSW* lpOP;
} OLEUIGNRLPROPSW, *POLEUIGNRLPROPSW, FAR* LPOLEUIGNRLPROPSW;
```

Members

cbStruct

Size of the structure in bytes. This field must be filled on input.

dwFlags

Currently no flags associated with this member. It should be set to 0 (zero).

dwReserved1[2]

Reserved for future use.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook* member during WM_INITDIALOG.

dwReserved2[3]

Reserved for future use.

lpOP

Used internally.

See Also

OleUIObjectProperties, **OLEUIOBJECTPROPS**

OLEUIINSERTOBJECT Quick Info

The OLEUIINSERTOBJECT structure contains information that the OLE User Interface Library uses to initialize the Insert Object dialog box, and space for the library to return information when the dialog box is dismissed.

```
typedef struct tagOLEUIINSERTOBJECT
{
// These IN fields are standard across all OLEUI dialog box functions.
    DWORD          cbStruct;
    DWORD          dwFlags;
    HWND           hWndOwner;
    LPCSTR         lpszCaption;
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    HINSTANCE      hInstance;
    LPCSTR         lpszTemplate;
    HRSRC          hResource;
    CLSID          clsid;

// Specifics for OLEUIINSERTOBJECT.
    LPTSTR         lpszFile;
    UINT           cchFile;
    UINT           cClsidExclude;
    LPCLSID        lpClsidExclude;
    IID            iid;

// Specific to create objects if flags say so
    DWORD            oleRender;
    LPFORMATETC      lpFormatEtc;
    LPOLECLIENTSITE  lpIOleClientSite;
    LPSTORAGE        lpIStorage;
    LPVOID FAR *     ppvObj;
    SCODE            sc;
    HGLOBAL          hMetaPict;
} OLEUIINSERTOBJECT, *POLEUIINSERTOBJECT, FAR *LPOLEUIINSERTOBJECT;
```

Members

cbStruct

Size of the structure in bytes. This field must be filled on input.

dwFlags

On input, specifies the initialization and creation flags. On exit, specifies the user's choices. It can be a combination of the following flags:

IOF_SHOWHELP

The dialog box will display a Help button.

IOF_SELECTCREATENEW

The Create New radio button will initially be checked. This cannot be used with IOF_SELECTCREATEFROMFILE.

IOF_SELECTCREATEFROMFILE

The Create From File radio button will initially be checked. This cannot be used with IOF_SELECTCREATENEW.

IOF_CHECKLINK

The Link check box will initially be checked.

IOF_CHECKDISPLAYASICON

The Display As Icon check box will initially be checked, the current icon will be displayed, and the Change Icon button will be enabled.

IOF_CREATENEWOBJECT

A new object should be created when the user selects OK to dismiss the dialog box and the Create New radio button was selected.

IOF_CREATEFILEOBJECT

A new object should be created from the specified file when the user selects OK to dismiss the dialog box and the Create From File radio button was selected.

IOF_CREATELINKOBJECT

A new linked object should be created when the user selects OK to dismiss the dialog box and the user checked the Link check box.

IOF_DISABLELINK

The Link check box will be disabled on initialization.

IOF_VERIFYSERVERSEXIST

The dialog box should validate the classes it adds to the listbox by ensuring that the server specified in the registration database exists. This is a significant performance factor.

IOF_DISABLEDISPLAYASICON

The Display As Icon check box will be disabled on initialization.

IOF_HIDECHANGEICON

The Change Icon button will be hidden in the Insert Object dialog box.

IOF_SHOWINSERTCONTROL

Displays the Insert Control radio button.

IOF_SELECTCREATECONTROL

Displays the Create Control radio button.

hWndOwner

Window that owns the dialog box. It should not be NULL.

lpszCaption

Pointer to a string to be used as the title of the dialog box. If NULL, then the library uses Insert Object.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook*

member. The library passes a pointer to the OLEUIINSERTOBJECT structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the *lCustData* member.

hInstance

Instance that contains a dialog box template specified by the *lpTemplateName* member.

lpszTemplate

Pointer to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the library's Insert Object dialog box template.

hResource

Customized template handle.

clsid

CLSID for class of the object to be inserted. Filled on output.

lpszFile

Pointer to the name of the file to be linked or embedded. Filled on output.

cchFile

Size of *lpszFile* buffer; will not exceed OLEUI_CCHPATHMAX.

cClsidExclude

Number of CLSIDs included in the *lpClsidExclude* list. Filled on input.

lpClsidExclude

Pointer to a list of CLSIDs to exclude from listing.

iid

Identifier of the requested interface. If **OleUIInsertObject** creates the object, then it will return a pointer to this interface. This parameter is ignored if **OleUIInsertObject** does not create the object.

oleRender

Rendering option. If **OleUIInsertObject** creates the object, then it selects the rendering option when it creates the object. This parameter is ignored if **OleUIInsertObject** does not create the object.

lpFormatEtc

Desired format. If **OleUIInsertObject** creates the object, then it selects the format when it creates the object. This parameter is ignored if **OleUIInsertObject** does not create the object.

lpIOleClientSite

Pointer to the client site to be used for the object. This parameter is ignored if **OleUIInsertObject** does not create the object.

lpIStorage

Pointer to the storage to be used for the object. This parameter is ignored if **OleUIInsertObject** does not create the object.

ppvObj

Indirect pointer to where the object is returned. This parameter is ignored if **OleUIInsertObject** does not create the object.

sc

Result of creation calls. This parameter is ignored if **OleUIInsertObject** does not create the object.

hMetaPict

MetafilePict structure containing the iconic aspect, if it wasn't placed in the object's cache.

See Also

OleUIInsertObject

OLEUILINKPROPS Quick Info

This structure is used to initialize the Link tab of the Object Properties dialog box. A reference to it is passed in as part of the OLEUIOBJECTPROPS structure to the **OleUIObjectProperties** function. This tab shows the location, update status, and update time for a link. It allows the user to change the source of the link, toggle its update status between automatic and manual update, open the source, force an update of the link, or break the link (convert it to a static picture).

```
// These IN fields are standard across all OLEUI property pages.

typedef struct tagOLEUILINKPROPSW
{
// These IN fields are standard across all OLEUI property pages.
    DWORD          cbStruct;
    DWORD          dwFlags;
    DWORD          dwReserved1[2];
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    DWORD          dwReserved2[3];

    struct tagOLEUIOBJECTPROPSW* lpOP;
} OLEUILINKPROPSW, *POLEUILINKPROPSW, FAR* LPOLEUILINKPROPSW;
```

Members

cbStruct

Size of the structure in bytes.

dwFlags

Contains in/out flags specific to the Links page.

dwReserved1[2]

Reserved for future use.

lpfnHook

Pointer to the hook callback (not used in this dialog box).

lCustData

Custom data to pass to hook (not used in this dialog box).

dwReserved2[3]

Reserved for future use.

lpOP

Used internally.

See Also

OleUIObjectProperties, **OLEUIOBJECTPROPS**

OLEUIOBJECTPROPS Quick Info

This structure is used to initialize the standard Object Properties dialog box. It contains references to interfaces used to gather information about the embedding or link, references to three structures that are used to initialize the default tabs—General (OLEUIGNRLPROPS), View (OLEUIVIEWPROPS), and Link (OLEUILINKPROPS), if appropriate—and a standard property-sheet extensibility interface that allows the caller to add additional custom property sheets to the dialog box.

```
typedef struct tagOLEUIOBJECTPROPS
{

// These IN fields are standard across all OLEUI property sheets.

    DWORD    cbStruct;
    DWORD    dwFlags;

// Standard PROPSHEETHEADER used for extensibility
    LPPROPSHEETHEADER        lpPS;

// Data which allows manipulation of the object
    DWORD    dwObject;
    LPOLEUIOBJINFO        lpObjInfo;

// Data which allows manipulation of the link
    DWORD    dwLink;
    LPOLEUILINKINFO        lpLinkInfo;

// Data specfic to each page
    LPOLEUIGNRLPROPS        lpGP;
    LPOLEUIVIEWPROPS        lpVP;
    LPOLEUILINKPROPS        lpLP;

} OLEUIOBJECTPROPS, *POLEUIOBJECTPROPS, FAR* LPOLEUIOBJECTPROPS;
```

Members

cbStruct

Size of the structure in bytes.

dwFlags

Contains in/out global flags for the property sheet.

OPF_OBJECTISLINK

Object is a link object and therefore has a link property page.

OPF_NOFILLDEFAULT

Do not fill in default values for the object.

OPF_SHOWHELP

Dialog box will display a Help button.

OPF_DISABLECONVERT

The Convert button will be disabled on the general property page.

lpPS

[in] Pointer to the standard property sheet header (PROPSHEETHEADER), used for extensibility.

//Data which allows manipulation of the object

dwObject

[in] Identifier for the object.

lpObjInfo

[in] Pointer to the interface to manipulate object.

//Data which allows manipulation of the link

dwLink;

[in] Container-defined unique 32-bit identifier for a single link. Containers can use the pointer to the link's container site for this value.

lpLinkInfo

[in] Pointer to the interface to manipulate link.

// Data specific to each page

lpGP

[in] Pointer to the general page data.

lpVP

[in] Pointer to the view page data.

lpLP

[in] Pointer to the link page data.

See Also

OleUIObjectProperties, **OLEUIGNRLPROPS**, **OLEUIVIEWPROPS**, **OLEUILINKPROPS**

OLEUIPASTEENTRY Quick Info

This structure is an array of OLEUIPASTEENTRY entries specified in the OLEUIPASTESPECIAL structure for the Paste Special dialog box. Each entry includes a FORMATETC which specifies the formats that are acceptable, a string that is to represent the format in the dialog box's listbox, a string to customize the result text of the dialog box, and a set of flags from the OLEUIPASTEFLAG enumeration. The flags indicate if the entry is valid for pasting only, linking only or both pasting and linking. If the entry is valid for linking, the flags indicate which link types are acceptable by OR'ing together the appropriate OLEUIPASTE_LINKTYPE<#> values. These values correspond to the array of link types as follows:

OLEUIPASTE_LINKTYPE1=arrLinkTypes[0]
OLEUIPASTE_LINKTYPE2=arrLinkTypes[1]
OLEUIPASTE_LINKTYPE3=arrLinkTypes[2]
OLEUIPASTE_LINKTYPE4=arrLinkTypes[3]
OLEUIPASTE_LINKTYPE5=arrLinkTypes[4]
OLEUIPASTE_LINKTYPE6=arrLinkTypes[5]
OLEUIPASTE_LINKTYPE7=arrLinkTypes[6]
OLEUIPASTE_LINKTYPE8=arrLinkTypes[7]

arrLinkTypes[] is an array of registered clipboard formats for linking. A maximum of eight link types are allowed.

```
typedef struct tagOLEUIPASTEENTRY
{
    FORMATETC   fmtetc;
    LPCSTR      lpstrFormatName;
    LPCSTR      lpstrResultText;
    DWORD       dwFlags;
    DWORD       dwScratchSpace;
} OLEUIPASTEENTRY, *POLEUIPASTEENTRY, FAR *LPOLEUIPASTEENTRY;
```

Members

fmtetc

Format that is acceptable. The Paste Special dialog box checks if this format is offered by the object on the clipboard and if so, offers it for selection to the user.

lpstrFormatName

Pointer to the string that represents the format to the user. Any %s in this string is replaced by the *FullUserTypeName* of the object on the clipboard and the resulting string is placed in the list box of the dialog box. Only one %s is allowed. The presence or absence of %s specifies whether the result-text is to indicate that data is being pasted or that an object that can be activated by an application is being pasted. If %s is present, the resulting text says that an object is being pasted. Otherwise, it says that data is being pasted.

lpstrResultText

Pointer to the string used to customize the resulting text of the dialog box when the user selects the format corresponding to this entry. Any %s in this string is replaced by the application name or *FullUserTypeName* of the object on the clipboard. Only one %s is allowed.

dwFlags

Values from OLEUIPASTEFLAG enumeration.

dwScratchSpace

Scratch space available to routines that loop through an **IEnumFORMATETC** to mark if the *PasteEntry* format is available. This field CAN be left uninitialized.

See Also

OLEUIPASTEFLAG, **OleUIPasteSpecial**, **OLEUIPASTESPECIAL**

OLEUIPASTESPECIAL Quick Info

The OLEUIPASTESPECIAL structure contains information that the OLE User Interface Library uses to initialize the Paste Special dialog box, as well as space for the library to return information when the dialog box is dismissed.

```
typedef struct tagOLEUIPASTESPECIAL
{
// These IN fields are standard across all OLEUI dialog box functions.
    DWORD          cbStruct;
    DWORD          dwFlags;
    HWND           hWndOwner;
    LPCSTR         lpszCaption;
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    HINSTANCE      hInstance;
    LPCSTR         lpszTemplate;
    HRSRC          hResource;

// Specifics for       OLEUIPASTESPECIAL.
    LPDATAOBJECT       lpSrcDataObj;
    LPOLEUIPASTEENTRY  arrPasteEntries;
    int                cPasteEntries;
    UINT FAR *         arrLinkTypes;
    int                cLinkTypes;
    UINT               cClsidExclude;
    LPCLSID            lpClsidExclude;
    int                nSelectedIndex;
    BOOL               fLink;
    HGLOBAL            hMetaPict;
    SIZEL              sizel;
} OLEUIPASTESPECIAL, *POLEUIPASTESPECIAL, FAR *LPOLEUIPASTESPECIAL;
```

Members

cbStruct

Size of the structure, in bytes. This field must be filled on input.

dwFlags

On input, *dwFlags* specifies the initialization and creation flags. On exit, it specifies the user's choices. It may be a combination of the following flags:

PSF_SHOWHELP

Dialog box will display a Help button.

PSF_SELECTPASTE

The Paste radio button will be selected at dialog box startup. This is the default, if PSF_SELECTPASTE or PSF_SELECTPASTELINK are not specified. Also, it specifies the state of the button on dialog termination. IN/OUT flag.

PSF_SELECTPASTELINK

The PasteLink radio button will be selected at dialog box startup. Also, specifies the state of the button on dialog termination. IN/OUT flag.

PSF_CHECKDISPLAYASICON

Whether the Display As Icon radio button was checked on dialog box termination. OUT flag.

PSF_DISABLEDISPLAYASICON

The Display As Icon check box will be disabled on initialization.

HIDECHANGEICON

Used to disable the change-icon button in the dialog box , which is available to users when they're pasting an OLE object by default. See STAYONCLIPBOARDCHANGE otherwise.

STAYONCLIPBOARDCHANGE

Used to tell the dialog box to stay up if the clipboard changes while the dialog box is up. If the user switches to another application and copies or cuts something, the dialog box will, by default, perform a cancel operation, which will remove the dialog box since the options it's in the middle of presenting to the user are no longer up-to-date with respect to what's really on the clipboard.

NOREFRESHDATAOBJECT

Used in conjunction with STAYONCLIPBOARDCHANGE (it doesn't do anything otherwise). If the clipboard changes while the dialog box is up and STAYONCLIPBOARDCHANGE is specified, then NOREFRESHDATAOBJECT indicates that the dialog box should NOT refresh the contents of the dialog box to reflect the new contents of the clipboard. This is useful if the application is using the paste-special dialog box on an IDataObject besides the one on the clipboard, for example, as part of a right-click drag-and-drop operation.

hWndOwner

Wwindow that owns the dialog box. It should not be NULL.

lpszCaption

Pointer to a string to be used as the title of the dialog box. If NULL, then the library uses Paste Special.

lpfnHook

Pointer to a hook function that processes messages intended for the dialog box. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in the library. The hook function must return a non-zero value to prevent the library's dialog box procedure from processing a message it has already processed.

lCustData

Application-defined data that the library passes to the hook function pointed to by the *lpfnHook* member. The library passes a pointer to the OLEUIPASTESPECIAL structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the *lCustData* member.

hInstance

Instance that contains a dialog box template specified by the *lpTemplateName* member.

lpszTemplate

Pointer to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the library's Paste Special dialog box template.

hResource

Customized template handle.

lpSrcDataObj

Pointer to the IDataObject* interface of the data object to be pasted (from the clipboard). This field is filled on input. If *lpSrcDataObj* is NULL when **OleUIPasteSpecial** is called, then **OleUIPasteSpecial** will attempt to retrieve a pointer to an IDataObject from the clipboard. If **OleUIPasteSpecial** succeeds, it is the caller's responsibility to free the IDataObject returned in *lpSrcDataObj*.

arrPasteEntries

The OLEUIPASTEENTRY array which specifies acceptable formats. This field is filled on input.

cPasteEntries

Number of OLEUIPASTEENTRY array entries. This field is filled on input.

arrLinkTypes

List of link types that are acceptable. Link types are referred to using OLEUIPASTEFLAG in arrPasteEntries. This field is filled on input.

cLinkTypes

Number of link types. This field is filled on input.

cClsidExclude

Number of CLSIDs in *lpClsidExclude*. This field is filled on input.

lpClsidExclude

Pointer to an array of CLSIDs to exclude from the list of available server objects for a Paste operation. Note that this does not affect Paste Link. An application can prevent embedding into itself by listing its own CLSID in this list. This field is filled on input.

nSelectedIndex

Index of *arrPasteEntries*[] that the user selected. This field is filled on output.

fLink

Whether Paste or Paste Link was selected by the user. This field is filled on output.

hMetaPict

Handle to the Metafile containing the icon and icon title selected by the user. This field is filled on output.

sizel

Size of object as displayed in its source, if the display aspect chosen by the user matches the aspect displayed in the source. If the user chooses a different aspect, then *sizel.cx* and *sizel.cy* are both set to zero. The size of the object as it is displayed in the source is retrieved from the ObjectDescriptor if *fLink* is FALSE and from the LinkSrcDescriptor if *fLink* is TRUE. This field is filled on output.

See Also

OleUIPasteSpecial, **OLEUIPASTEENTRY**, **OLEUIPASTEFLAG**

OLEUIVIEWPROPS Quick Info

This structure is used to initialize the View tab of the Object properties dialog box. A reference to it is passed in as part of the **OLEUIOBJECTPROPS** structure to the **OleUIObjectProperties** function. This tab allows the user to toggle between "content" and "iconic" views of the object, and change its scaling within the container. It also allows the user to tunnel to the change icon dialog box when the object is being displayed iconically.

```
// These IN fields are standard across all OLEUI property pages.

typedef struct tagOLEUIVIEWPROPSA
{
// These IN fields are standard across all OLEUI property pages.
    DWORD          cbStruct;
    DWORD          dwFlags;
    DWORD          dwReserved1[2];
    LPFNOLEUIHOOK  lpfnHook;
    LPARAM         lCustData;
    DWORD          dwReserved2[3];

struct tagOLEUIOBJECTPROPS* lpOP;

    int       nScaleMin;
    int       nScaleMax;

} OLEUIVIEWPROPSA, *POLEUIVIEWPROPSA, FAR* LPOLEUIVIEWPROPSA;
```

Members

cbStruct

Size of the structure in bytes.

dwFlags

IN-OUT: flags specific to view page

VPF_SELECTRELATIVE

[in] Relative to origin.

VPF_DISABLERELATIVE

[in] Disable relative to origin.

VPF_DISABLESCALE

[in] Disable scale option.

dwReserved1[2]

Reserved for future use.

lpfnHook

Pointer to hook callback (not used in this dialog box).

lCustData

Custom data to pass to hook (not used in this dialog box).

dwReserved2[3];

Reserved for future use.

lpOP;

Used internally.

nScaleMin

Minimum value for the scale range.

nScaleMax

Maximum value for the scale range.

See Also

OleUIObjectProperties, **OLEUIOBJECTPROPS**

OLEVERB Quick Info

The **OLEVERB** structure defines a verb that an object supports. The **IOleObject::EnumVerbs** method creates an enumerator that can enumerate these structures for an object, and supplies a pointer to the enumerator's **IEnumOLEVERB**.

```
typedef struct tagOLEVERB
{
    LONG      lVerb;
    LPWSTR    lpszVerbName;
    DWORD     fuFlags;
    DWORD     grfAttribs;
} OLEVERB, * LPOLEVERB;
```

Members

lVerb

Integer identifier associated with this verb.

lpszVerbName

Pointer to a string that contains the verb's name.

fuFlags

In Windows, a group of flags taken from the flag constants beginning with MF_ defined in **AppendMenu**. Containers should use these flags in building an object's verb menu. All Flags defined in **AppendMenu** are supported except for MF_BITMAP, MF_OWNERDRAW, and MF_POPUP.

grfAttribs

Combination of the verb attributes in the **OLEVERBATTRIB** enumeration.

See Also

IEnumOLEVERB, **IOleObject::EnumVerbs**

PICTDESC Quick Info

The **PICTDESC** structure contains parameters to create a picture object through the **OleCreatePictureIndirect** function.

```
typedef struct tagPICTDESC
{
    UINT cbSizeOfStruct;
    UINT picType;
    union
    {
        struct
        {
            HBITMAP hbitmap;
            HPALETTE hpal;
        } bmp;
        struct
        {
            HMETAFILE hmeta;
            int xExt;
            int yExt;
        } wmf;
        struct
        {
            HICON hicon;
        } icon;
        struct
        {
            HENHMETAFILE hemf;
        } emf;
    } ;
} PICTDESC;
```

Members

cbSizeOfStruct

Size of the **PICTDESC** structure.

picType

Type of picture described by this structure, which can be any value from the **PICTYPE** enumeration.

bmp

Structure containing bitmap information if *picType* is PICTYPE_BITMAP.

bmp.hbitmap

The HBITMAP identifying the bitmap assigned to the picture object.

bmp.hpal

The HPALETTE identifying the color palette for the bitmap.

wmf

Structure containing metafile information if *picType* is PICTYPE_METAFILE.

wmf.hmeta

The HMETAFILE handle identifying the metafile assigned to the picture object.

wmf.xExt

Horizontal extent of the metafile in HIMETRIC units.

wmf.yExt

Vertical extent of the metafile in HIMETRIC units.

icon

Identifies a structure containing icon information if *picType* is PICTYPE_ICON.

icon.hicon

The HICON identifying the icon assigned to the picture object.

emf

Structure containing enhanced metafile information if *picType* is PICTYPE_ENHMETAFILE.

emf.hemf

The HENHMETAFILE identifying the enhanced metafile to assign to the picture object.

See Also

OleCreatePictureIndirect, **PICTYPE**

POINTF Quick Info

The **POINTF** structure is used in the **IOleControlSite::TransformCoords** method to convert between container units, expressed in floating point, and control units, expressed in HIMETRIC. The **POINTF** structure specifically holds the floating point container units. Controls do not attempt to interpret either value in the structure.

```
typedef struct tagPOINTF
{
    float x;
    float y;
} POINTF;
```

Members

x

The x-coordinates of the point in units that the container finds convenient.

y

The y coordinates of the point in units that the container finds convenient.

See Also

IControlOleSite::TransformCoords

PROPPAGEINFO Quick Info

The **PROPPAGEINFO** structure contains parameters used to describe a property page to a property frame. A property page fills a caller-provided structure in the **IPropertyPage::GetPageInfo** method.

The *pszTitle*, *pszDocString*, and the *pszHelpFile* members specified in this structure should contain text sensitive to the locale obtained through **IPropertyPageSite::GetLocaleID**.

```
typedef struct tagPROPPAGEINFO
{
    ULONG     cb;
    LPOLESTR  pszTitle;
    SIZE      size;
    LPOLESTR  pszDocString;
    LPOLESTR  pszHelpFile;
    DWORD     dwHelpContext;
} PROPPAGEINFO;
```

Members

cb

Size of the **PROPPAGEINFO** structure.

pszTitle

Pointer to the string that appears in the tab for this page. The string must be allocated with **CoTaskMemAlloc**. The caller of **IPropertyPage::GetPageInfo** is responsible for freeing the memory with **CoTaskMemFree**.

size

Required dimensions of the page's dialog box, in pixels.

pszDocString

Pointer to a text string describing the page, which can be displayed in the property sheet dialog box (current frame implementation doesn't use this field). The text must be allocated with **CoTaskMemAlloc**. The caller of **IPropertyPage::GetPageInfo** is responsible for freeing the memory with **CoTaskMemFree**.

pszHelpFile

Pointer to the simple name of the help file that describes this property page used instead of implementing **IPropertyPage::Help**. When the user presses Help, the **Help** method is normally called. If that method fails, the frame will open the help system with this help file (prefixed with the value of the HelpDir key in the property page's registry entries under its CLSID) and will instruct the help system to display the context described by the *dwHelpContext* field. The path must be allocated with **CoTaskMemAlloc**. The caller of **IPropertyPage::GetPageInfo** is responsible for freeing the memory with **CoTaskMemFree**.

dwHelpContext

Context identifier for the help topic within *pszHelpFile* that describes this page.

See Also

CoTaskMemAlloc, **CoTaskMemFree**, **IPropertyPageSite::GetLocaleID**, **IPropertyPage::GetPageInfo**, **IPropertyPage::Help**

PROPSPEC Quick Info

The PROPSPEC structure is used by many of the methods of **IPropertyStorage** to specify a property either by its property identifier or the associated string name. The structure and related definitions are defined as follows in the header files:

```
const ULONG     PRSPEC_LPWSTR = 0
const ULONG     PRSPEC_PROPID = 1

typedef ULONG     PROPID

typedef struct tagPROPSPEC
{
    ULONG ulKind;         // PRSPEC_LPWSTR or PRSPEC_PROPID
    union
    {
    PROPID      propid;
    LPOLESTR    lpwstr;
    }
} PROPSPEC
```

Members

ulKind

If *ulKind* is set to PRSPEC_LPWSTR, *lpwstr* is used and set to a string name. If *ulKind* is set to PRSPEC_PROPID, *propid* is used and set to a property identifier value.

propid

Specifies the value of the property identifier. Use either this value or the following *lpwstr*, not both.

lpwstr

Specifies the string name of the property as a null-terminated Unicode string.

Remarks

String names are optional and can be assigned to a set of properties when the property is created with a call to **IPropertyStorage::WriteMultiple**, or later, with a call to **IPropertyStorage::WritePropertyNames**.

See Also

IPropertyStorage

PROPVARIANT Quick Info

The PROPVARIANT structure is used in most of the methods of **IPropertyStorage** to define the type tag and the value of a property in a property set. There are five members. The first, the value type tag, and the last, the value of the property, are significant. The middle three are reserved for future use. The PROPVARIANT structure is defined as follows:

Note The *bool* member in previous definitions of this structure has been renamed to *boolVal*, since some compilers now recognize *bool* as a keyword.

```
struct PROPVARIANT{
    VARTYPE        vt;          // value type tag
    WORD           wReserved1;
    WORD           wReserved2;
    WORD           wReserved3;
    union {
    // none                      // VT_EMPTY, VT_NULL, VT_ILLEGAL
    unsigned char  bVal;        // VT_UI1
    short          iVal;        // VT_I2
    USHORT         uiVal;       // VT_UI2
    long           lVal;        // VT_I4
    ULONG          ulVal;       // VT_UI4
    LARGE_INTEGER  hVal;        // VT_I8
    ULARGE_INTEGER uhVal;       // VT_UI8
    float          fltVal;      // VT_R4
    double         dblVal;      // VT_R8
    CY             cyVal;       // VT_CY
    DATE           date;        // VT_DATE
    BSTR           bstrVal;     // VT_BSTR
    VARIANT_BOOL   boolVal;        // VT_BOOL
    SCODE          scode;       // VT_ERROR
    FILETIME       filetime;    // VT_FILETIME
    LPSTR          pszVal;      // VT_LPSTR     // string in the current
system Ansi code page
    LPWSTR         pwszVal;     // VT_LPWSTR    // string in Unicode
    CLSID*         puuid;       // VT_CLSID
    CLIPDATA*      pclipdata;   // VT_CF

    BLOB           blob;        // VT_BLOB, VT_BLOBOBJECT
    IStream*       pStream;     // VT_STREAM, VT_STREAMED_OBJECT
    IStorage*      pStorage;    // VT_STORAGE, VT_STORED_OBJECT

    CAUB           caub;         // VT_VECTOR | VT_UI1
    CAI            cai;          // VT_VECTOR | VT_I2
    CAUI           caui;         // VT_VECTOR | VT_UI2
    CAL            cal;          // VT_VECTOR | VT_I4
    CAUL           caul;         // VT_VECTOR | VT_UI4
    CAH            cah;          // VT_VECTOR | VT_I8
    CAUH           cauh;         // VT_VECTOR | VT_UI8
    CAFLT          caflt;        // VT_VECTOR | VT_R4
    CADBL          cadbl;        // VT_VECTOR | VT_R8
    CACY           cacy;         // VT_VECTOR | VT_CY
```

```
    CADATE          cadate;       // VT_VECTOR | VT_DATE
    CABSTR          cabstr;       // VT_VECTOR | VT_BSTR
    CABOOL          cabool;       // VT_VECTOR | VT_BOOL
    CASCODE         cascode;      // VT_VECTOR | VT_ERROR
    CALPSTR         calpstr;      // VT_VECTOR | VT_LPSTR
    CALPWSTR        calpwstr;     // VT_VECTOR | VT_LPWSTR
    CAFILETIME      cafiletime;   // VT_VECTOR | VT_FILETIME
    CACLSID         cauuid;       // VT_VECTOR | VT_CLSID
    CACLIPDATA      caclipdata;   // VT_VECTOR | VT_CF
    CAPROPVARIANT   capropvar;    // VT_VECTOR | VT_VARIANT
    }} PROPVARIANT
```

Remarks

PROPVARIANT is the fundamental data type by which property values are read and written through the **IPropertyStorage** interface.

The data type PROPVARIANT is related to the data type VARIANT, defined as part of Automation in OLE2 and defined in the Win32 SDK header file oleauto.h. Several definitions are reused from Automation, as follows:

```
typedef struct  tagCY {
    unsigned long      Lo;
    long               Hi;
    } CY

typedef CY             CURRENCY;
typedef short          VARIANT_BOOL;
typedef unsigned short VARTYPE;
typedef double         DATE;
typedef OLECHAR*       BSTR;

typedef struct         tagCLIPDATA {
    ULONG              cbSize;  //Includes sizeof(ulClipFmt)
    long               ulClipFmt;
    BYTE*              pClipData;
    } CLIPDATA
```

In addition, several new data types that define counted arrays of other data types are required. The data types of all counted arrays begin with the letters **CA** (such as **CAUB**) and have an ORed *vt* value. The counted array structure has the following form (where *name* is the specific name of the counted array):

```
#define TYPEDEF_CA(type, name)

    typedef struct tag ## name {\
        ULONG cElems;\
        type *pElems;\
        } name
```

Propvariant Type	Code	Propvariant Member	Value Representation
VT_EMPTY	0	None	A property with a type indicator of VT_EMPTY has no data associated with it; that is, the size of the value is zero.
VT_NULL	1	None	This is like a pointer to NULL.

VT_UI1	17	bVal	1-byte unsigned integer
VT_I2	2	iVal	Two bytes representing a 2-byte signed integer value.
VT_UI2	18	uiVal	2-byte unsigned integer
VT_I4	3	lVal	4-byte signed integer value
VT_UI4	19	ulVal	4-byte unsigned integer
VT_I8	20	hVal	8-byte signed integer
VT_UI8	21	uhVal	8-byte unsigned integer
VT_R4	4	fltVal	32-bit IEEE floating point value
VT_R8	5	dblVal	64-bit IEEE floating point value
VT_CY	6	cyVal	8-byte two's complement integer (scaled by 10,000). This type is commonly used for currency amounts.
VT_DATE	7	date	A 64-bit floating point number representing the number of days (not seconds) since December 31, 1899. For example, January 1, 1900 is 2.0, January 2, 1900 is 3.0, and so on). This is stored in the same representation as VT_R8.
VT_BSTR	8	bstrVal	Pointer to a null terminated Unicode string. The string is immediately preceded by a DWORD representing the byte count, but *bstrVal* points past this DWORD to the first character of the string. BSTRs must be allocated and freed using the OLE Automation SysAllocString and SysFreeString calls.
VT_BOOL	11	boolVal (bool in earlier designs)	Boolean value, a WORD containing 0 (false) or -1 (true).
VT_ERROR	10	scode	A DWORD containing a status code.
VT_FILETIME	64	filetime	64-bit **FILETIME** structure as defined by Win32. It is recommended that all times be stored in Universal Coordinate Time (UTC).
VT_LPSTR	30	pszVal	Pointer to a null terminated ANSI string in the system default code page.
VT_LPWSTR	31	pwszVal	Pointer to a null terminated Unicode string in the user's default locale.
VT_CLSID	72	puuid	Pointer to a CLSID (or other GUID).
VT_CF	71	pclipdata	Pointer to a CLIPDATA structure, described above.
VT_BLOB	65	blob	DWORD count of bytes, followed by that many bytes of data. The byte count does not include the four bytes for the length of the count itself; an empty BLOB would have a count of zero, followed by zero bytes. This is similar to VT_BSTR but does not guarantee a null byte at the end of the data.
VT_BLOBOBJECT	70	blob	A BLOB containing a serialized object in

			the same representation as would appear in a VT_STREAMED_OBJECT. That is, a DWORD byte count (where the byte count does not include the size of itself) which is in the format of a class identifier followed by initialization data for that class. The only significant difference between VT_BLOB_OBJECT and VT_STREAMED_OBJECT is that the former does not have the system-level storage overhead that the latter would have, and is therefore more suitable for scenarios involving numbers of small objects.
VT_STREAM	66	pStream	Pointer to an IStream interface, representing a stream which is a sibling to the "Contents" stream.
VT_STREAMED_OBJECT	68	pStream	As in VT_STREAM, but indicates that the stream contains a serialized object, which is a CLSID followed by initialization data for the class. The stream is a sibling to the Contents stream that contains the property set.
VT_STORAGE	67	pStorage	Pointer to an IStorage interface, representing a storage object that is a sibling to the "Contents" stream.
VT_STORED_OBJECT	69	pStorage	As in VT_STORAGE, but indicates that the designated IStorage contains a loadable object.
VT_VECTOR	0x1000	ca*	If the type indicator is one of the simple propvariant types ORed with this one, the value is one of the counted array values. This is a DWORD count of elements, followed by that many repetitions of the value. For example, a type indicator of VT_LPSTR|VT_VECTOR has a DWORD element count, a DWORD byte count, the first string data, padding bytes for 32-bit alignment (see below), a DWORD byte count, the second string data, and so on. Nonsimple types cannot be ORed with VT_VECTOR. These types are VT_STREAM, VT_STREAM_OBJECT, VT_STORAGE, VT_STORAGE_OBJECT. VT_BLOB and VT_BLOB_OBJECT types also cannot be ORed with VT_VECTOR.
VT_VARIANT	12	capropvar	A DWORD type indicator followed by the corresponding value. VT_VARIANT can be used only with VT_VECTOR.
VT_TYPEMASK	0xFFF		Used as a mask for VT_VECTOR and

other modifiers to extract the raw VT value.

Clipboard format identifiers, stored with the tag VT_CF, use one of five different representations (identified in the *ulClipFmt* member of the **CLIPDATA** structure):

***ulClipFmt* Value**	***pClipData* value**
-1L	a DWORD containing a built-in Windows clipboard format value.
-2L	a DWORD containing a Macintosh clipboard format value.
-3L	a GUID containing a format identifier (rarely used).
any positive value	a null-terminated string containing a Windows clipboard format name, one suitable for passing to **RegisterClipboardFormat**. The code page used for characters in the string is per the code page indicator. The "positive value" here is the length of the string, including the null byte at the end.
0L	no data (rarely used)

Within a vector of values, each repetition of a value is to be aligned to 32-bit boundaries. The exception to this rule is scalar types which are less than 32 bits: VT_UI1, VT_12, VT_U12, and VT_BOOL. Vectors of these values are packed.

Therefore, a value with type tag VT_I2 | VT_VECTOR would be a DWORD element count, followed by a sequence of packed 2-byte integers with *no* padding between them.

However, a value with type tag VT_LPSTR | VT_VECTOR would be a DWORD element count, followed by a sequence of (DWORD *cch*, char *rgch*[]) strings, each of which may be followed by null padding to round to a 32-bit boundary.

QACONTAINER Quick Info

The **QACONTAINER** structure is used in **IQuickActivate::QuickActivate** to specify container information.

```
typedef struct tagQACONTAINER
{
    ULONG                   cbSize;
    IOleClientSite*         pClientSite;
    IAdviseSinkEx*          pAdviseSink;
    IPropertyNotifySink*    pPropertyNotifySink;
    IUnknown*               pUnkEventSink;
    DWORD                   dwAmbientFlags;
    OLE_COLOR               colorFore;
    OLE_COLOR               colorBack;
    IFont*                  pFont;
    IOleUndoManager*        pUndoMgr;
    DWORD                   dwAppearance;
    LONG                    lcid;
    HPALETTE                hpal;
    struct IBindHost*       pBindHost;
}    QACONTAINER;
```

Members

cbsize

Specifies the size of the structure in bytes.

pClientSite

Pointer to an **IOleClientSite** interface in the container.

pAdviseSink

Pointer to an **IAdviseSinkEx** interface in the container.

pPropertyNotifySink

Pointer to an **IPropertyNotifySink** interface in the container.

pUnkEventSink

Pointer to an **IUnknown** interface on the container's sink object.

dwAmbientFlags

Specifies a number of ambient properties supplied by the container using values from the **QACONTAINERFLAGS** enumeration.

colorFore

Specifies ForeColor, an ambient property supplied by the container with a DISPID = -704.

colorBack

Specifies BackColor, an ambient property supplied by the container with a DISPID = -701.

pFont

Specifies Font, an ambient property supplied by the container with a DISPID = -703.

pUndoMgr

Pointer to an **IOleUndoManager** interface in the container.

dwAppearance

Specifies Appearance, an ambient property supplied by the container with a DISPID = -716.

lcid

Specifies LocaleIdentifier, an ambient property supplied by the container with a DISPID = -705.

hPal

Specifies Palette, an ambient property supplied by the container with a DISPID = -726.

pBindHost

Pointer to an **IBindHost** interface in the container.

Remarks

If an interface pointer in the **QACONTAINER** structure is NULL it does not indicate that the interface is not supported. In this situation, the control should use **QueryInterface** to obtain the interface pointer in the standard manner.

See Also

IQuickActivate::QuickActivate, **QACONTAINERFLAGS**

QACONTROL Quick Info

The **QACONTROL** structure is used in **IQuickActivate::QuickActivate** to specify control information.

```
typedef struct tagQACONTROL
{
    ULONG     cbSize;
    DWORD     dwMiscStatus;
    DWORD     dwViewStatus;
    DWORD     dwEventCookie;
    DWORD     dwPropNotifyCookie;
    DWORD     dwPointerActivationPolicy;
}QACONTROL;
```

Members

cbsize

Size of the structure in bytes.

dwMiscStatus

Specifies the control's miscellaneous status bits that can also be returned by **IOleObject::GetMiscStatus**. See **OLEMISC** for more information.

dwViewStatus

Specifies the control's view status that can also be returned by **IViewObjectEx::GetViewStatus**. See **VIEWSTATUS** for more information.

dwEventCookie

Unique identifier for control-defined events.

dwPropNotifyCookie

Unique identifier for control-defined properties.

dwPointerActivationPolicy

Specifies the control's activation policy that can also be returned by **IPointerInactive::GetActivationPolicy**. If all the bits of *dwPointerActivationPolicy* are set, then the **IPointerInactive** interface may not be supported. The container should **QueryInterface** to obtain the interface pointer in the standard manner.

See Also

IQuickActivate::QuickActivate

RemSNB Quick Info

The **RemSNB** structure is used for marshaling the **SNB** data type.

Defined in the **IStorage** interface (*storag.idl*).

```
typedef struct tagRemSNB {
    unsigned                long     ulCntStr;
    unsigned                long     ulCntChar;
    [size_is(ulCntChar)]    wchar_t  rgString[];
} RemSNB;
typedef [transmit_as(RemSNB)] wchar_t  **SNB;
```

Members

ulCntStr

Number of strings in the *rgString* buffer.

ulCntChar

Size in bytes of the *rgString* buffer.

rgString

Pointer to an array of bytes containing the stream name strings from the SNB.

See Also

IStorage

SNB Quick Info

A string name block (SNB) is a pointer to an array of pointers to strings, that ends in a NULL pointer. String name blocks are used by the **IStorage** interface and by function calls that open storage objects. The strings point to contained storage objects or streams that are to be excluded in the open calls.

```
typedef OLESTR **SNB
```

Remarks

The SNB should be created by allocating a contiguous block of memory in which the pointers to strings are followed by a NULL pointer, which is then followed by the actual strings.

The marshaling of a string name block is based on the assumption that the SNB passed in was created this way. Although it could be stored in other ways, the SNB created in this manner has the advantage of requiring only one allocation operation and one freeing of memory for all the strings.

See Also

IStorage

SOLE_AUTHENTICATION_SERVICE Quick Info

Identifies an authentication service. This structure is retrieved through a call to **CoQueryAuthenticationServices**, and passed in to **CoInitializeSecurity**.

```
typedef struct tagSOLE_AUTHENTICATION_SERVICE {
    DWORD       dwAuthnSvc;
    DWORD       dwAuthzSvc;
    OLECHAR*    pPrincipalName;
    HRESULT     hr;
} SOLE_AUTHENTICATION_SERVICE;
```

Members

dwAuthnSvc

The *authentication* service. It may contain a single value taken from the list of **RPC_C_AUTHN_*xxx*** constants defined in **rpcdce.h**. RPC_C_AUTHN_NONE turns off authentication. On Win32, RPC_C_AUTHN_DEFAULT causes COM to use the RPC_C_AUTHN_WINNT authentication.

dwAuthzSvc

The *authorization* service. It may contain a single value taken from the list of **RPC_C_AUTHZ_*xxx*** constants defined in rpcdce.h. The validity and trustworthiness of authorization data, like any application data, depends on the authentication service and authentication level selected. This parameter is ignored when using the RPC_C_AUTHN_WINNT authentication service.

pPrincipalName

Principal name to be used with the authentication service. If the principal name is NULL, COM assumes the current user identifier. A NULL principal name is allowed for NT LM SSP and *kerberos* authentication services, but may not work for other authentication services.

hr

When used in **CoInitializeSecurity**, set on return to indicate the status of the call to register the authentication services.

See Also

RPC_C_AUTHN_xxx, **RPC_C_AUTHZ_xxx**, **CoInitializeSecurity**

STATDATA Quick Info

The **STATDATA** structure is the data structure used to specify each advisory connection. It is used for enumerating current advisory connections. It holds data returned by the **IEnumSTATDATA** enumerator. This enumerator interface is returned by **IDataObject:DAdvise**. Each advisory connection is specified by a unique **STATDATA** structure.

```
typedef struct tagSTATDATA
{
    FORMATETC     formatetc;
    DWORD         grfAdvf;
    IAdviseSink*  pAdvSink;
    DWORD         dwConnection;
} STATDATA;
```

Members

formatetc

The **FORMATETC** structure for the data of interest to the advise sink. The advise sink receives notification of changes to the data specified by this **FORMATETC** structure.

grfAdvf

The **ADVF** enumeration value that determines when the advisory sink is notified of changes in the data.

pAdvSink

The pointer for the **IAdviseSink** interface that will receive change notifications.

dwConnection

The token that uniquely identifies the advisory connection. This token is returned by the method that sets up the advisory connection.

See Also

IEnumSTATDATA

STATPROPSETSTG Quick Info

Contains information about a property set. To get this information, call **IPropertyStorage::Stat**, which fills in a buffer containing the information describing the current property set. To enumerate the STATPROPSETSTG structures for the property sets in the current property set storage, call **IPropertySetStorage::Enum** to get a pointer to an enumerator. You can then call the enumeration methods of the **IEnumSTATPROPSETSTG** interface on the enumerator. The structure is defined as follows:

```
typedef struct tagSTATPROPSETSTG {
    FMTID       fmtid;
    CLSID       clsid;
    DWORD       grfFlags;
    FILETIME    mtime;
    FILETIME    ctime;
    FILETIME    atime;
    } STATPROPSETSTG
```

Members

fmtid

Format identifier of the current property set.

clsid

The CLSID associated with this property set.

grfFlags

Flag values of the property set, as specified in **IPropertySetStorage::Create**.

mtime

Time in Universal Coordinated Time (UTC) that the property set was last modified.

ctime

Time in UTC at which this property set was created.

atime

Time in UCT at which this property set was last accessed.

See Also

IPropertySetStorage::Create, **IEnumSTATPROPSETSTG**, **IPropertyStorage::Stat**, FILETIME structure

STATPROPSTG Quick Info

Each STATPROPSTG structure contains information about a single property in a property set. This information is the property identifier and type tag, and the optional string name that may be associated with the property.

IPropertyStorage::Enum supplies a pointer to the **IEnumSTATPROPSTG** interface on an enumerator object that can be used to enumerate through the STATPROPSTG structures for the properties in the current property set. STATPROPSTG is defined as follows:

```
typedef struct    tagSTATPROPSTG {
    LPWSTR        lpwstrName;
    PROPID        propid;
    VARTYPE       vt;
} STATPROPSTG
```

Members

lpwstrName

Wide-character string containing the optional string name that can be associated with the property. May be NULL. This member must be freed using **CoTaskMemFree**.

propid

A 32-bit identifier that uniquely identifies the property within the property set. All properties within property sets must have unique property identifiers.

vt

Type of the property.

See Also

IPropertyStorage::Enum, **IEnumSTATPROPSTG**

STATSTG Quick Info

The **STATSTG** structure contains statistical information about an open storage, stream, or byte array object. This structure is used in the **IEnumSTATSTG**, **ILockBytes**, **IStorage**, and **IStream** interfaces.

```
typedef struct tagSTATSTG
{
    LPWSTR          pwcsName;
    DWORD           type;
    ULARGE_INTEGER  cbSize;
    FILETIME        mtime;
    FILETIME        ctime;
    FILETIME        atime;
    DWORD           grfMode;
    DWORD           grfLocksSupported;
    CLSID           clsid;
    DWORD           grfStateBits;
    DWORD           reserved;
} STATSTG;
```

Members

pwcsName

Points to a NULL-terminated string containing the name. Space for this string is allocated by the method called and freed by the caller (refer to **CoTaskMemFree**). You can specify not to return this member by specifying the STATFLAG_NONAME value when you call a method that returns a **STATSTG** structure, except for calls to **IEnumSTATSTG::Next,** which provides no way to specify this value.

type

Indicates the type of storage object. This is one of the values from the **STGTY** enumeration.

cbSize

Specifies the size in bytes of the stream or byte array.

mtime

Indicates the last modification time for this storage, stream, or byte array.

ctime

Indicates the creation time for this storage, stream, or byte array.

atime

Indicates the last access time for this storage, stream or byte array.

grfMode

Indicates the access mode specified when the object was opened. This member is only valid in calls to **Stat** methods.

grfLocksSupported

Indicates the types of region locking supported by the stream or byte array. See the **LOCKTYPES** enumeration for the values available. This member is not used for storage objects.

clsid

Indicates the class identifier for the storage object; set to CLSID_NULL for new storage objects. This member is not used for streams or byte arrays.

grfStateBits

Indicates the current state bits of the storage object, that is, the value most recently set by the **IStorage::SetStateBits** method. This member is not valid for streams or byte arrays.

dwStgFmt

Indicates the format of the storage object. This is one of the values from the **STGFMT** enumeration.

See Also

IStorage::SetElementTimes

STORAGELAYOUT

The **STORAGELAYOUT** structure describes a single block of data, including its name, location, and length. To optimize a compound file, an application or layout tool passes an array of **StorageLayout** structures in a call to **ILayoutStorage::LayoutScript**.

```
typedef struct tagSTORAGELAYOUT
{
    DWORD           LayoutType;
    OLECHAR*        pwcsElementName;
    LARGE_INTEGER   cOffset;
    LARGE_INTEGER   cBytes;
} STORAGELAYOUT;
```

Members

LayoutType

The type of element to be written. Values are taken from the **STGTY** enumeration. STGTY_STREAM means read the block of data named by *pwcsElementName*. STGTY_STORAGE means open the storage specified in *pwcsElementName*. STGTY_REPEAT is used in multimedia applications to interlace audio, video, text, and other elements. An opening STGTY_REPEAT elements means that the elements that follow are to be repeated a specified number of times. The closing STGTY_REPEAT element marks the end of those elements that are to be repeated. Nested STGY_REPEAT pairs are permitted.

pwcsElementName

The name of the storage or stream. If the element is a substorage or embedded object, the fully qualified storage path must be specified; for example:RootStorageName\SubStorageName\Substream.

cOffset

Where *LayoutType* is STGTY_STREAM, this flag specifies the beginning offset into the steam named in *pwscElementName*.

Where *LayoutType* is STGTY_STORAGE, this flag should be set to zero.

Where *LayoutType* is STGTY_REPEAT, this flag should be set to zero.

cBytes

Length in bytes of the data block named in *pwcsElementName*.

Where *LayoutType* is STGTY_STREAM, *cBytes* specifies the number of bytes to read at offset *cOffset* from the stream nemaed in *pwcsElementName*.

Where *LayoutType* is STGTY_STORAGE, this flag is ignored.

Where *LayoutType* is STGTY_REPEAT, a positive *cBytes* specifies the beginning of a repeat block. STGTY_REPEAT with zero *cBytes* marks the end of a repeat block.

A beginning block value of STG_TOEND specifies that elements in a following block are to be repeated after each stream has been completely read.

An array of **StorageLayout** structures might appear as follows:

```
StorageLayout arrScript[]=
    // Read first 2k of "WordDocument" stream
    {STGTY_STREAM,L"WordDocument",{0,0},{0,2048}},
```

```
    //Test if "ObjectPool\88112233" storage exists
    {STGTY_STORAGE,L"ObjectPool\\88112233",{0,0},{0,0}},

    //Read 2k at offset 1048 of "WordDocument" stream
    {STGTY_STREAM,L"WordDocument",{0,10480},{0,2048}},

    //Interlace "Audio", "Video", and "Caption" streams
    {STGTY_REPEAT,NULL,0,STG_TOEND},
        {STGTY_STREAM,L"Audio", {0,0},{0,2048}},    // 2k of Audio
        {STGTY_STREAM,L"Video", {0,0},{0,65536}},   // 64k of Video
        {STGTY_STREAM,L"Caption", {0,0},{0,128}},   // 128b of text
    {STGTY_REPEAT,NULL, {0,0},{0,0}}
};
```

Note The parameters cOffest and cBytes are LARGE_INTEGER structures and they must be represented as a structure with {LARGE_INTEGER} or {DWORD lowpart, LONG highpart}.

See Also

ILayoutStorage::LayoutScript

STGMEDIUM Quick Info

The **STGMEDIUM** structure is a generalized global memory handle used for data transfer operations by the **IAdviseSink**, **IDataObject**, and **IOleCache** interfaces.

```
typedef struct tagSTGMEDIUM
{
    DWORD tymed;
    [switch_type(DWORD), switch_is((DWORD) tymed)]
    union {
        [case(TYMED_GDI)]       HBITMAP        hBitmap;
        [case(TYMED_MFPICT)]    HMETAFILEPICT  hMetafilePict;
        [case(TYMED_ENHMF)]     HENHMETAFILE   hEnhMetaFile;
        [case(TYMED_HGLOBAL)]   HGLOBAL        hGlobal;
        [case(TYMED_FILE)]      LPWSTR         lpszFileName;
        [case(TYMED_ISTREAM)]   IStream        *pstm;
        [case(TYMED_ISTORAGE)]  IStorage       *pstg;
        [default] ;
    };
    [unique] IUnknown *pUnkForRelease;
}STGMEDIUM;
typedef STGMEDIUM *LPSTGMEDIUM;
```

Members

tymed

Type of storage medium. The marshaling and unmarshaling routines use this value to determine which union member was used. This value must be one of the elements of the **TYMED** enumeration.

union member

Handle, string, or interface pointer that the receiving process can use to access the data being transferred. If *tymed* is TYMED_NULL, the union member is undefined; otherwise, it is one of the following:

hBitmap

Bitmap handle. The *tymed* member is TYMED_GDI.

hMetafilePict

Metafile handle. The *tymed* member is TYMED_MFPICT.

hEnhMetaFile

Enhanced metafile handle. The *tymed* member is TYMED_ENHMF.

hGlobal

Global memory handle. The *tymed* member is TYMED_HGLOBAL.

lpszFileName

Pointer to the path of a disk file that contains the data. The *tymed* member is TYMED_FILE.

pstm

Pointer to an **IStream** interface. The *tymed* member is TYMED_ISTREAM.

pstg

Pointer to an **IStorage** interface. The *tymed* member is TYMED_ISTORAGE.

pUnkForRelease

Pointer to an interface instance that allows the sending process to control the way the storage is released when the receiving process calls the **ReleaseStgMedium** function. If *pUnkForRelease* is NULL, **ReleaseStgMedium** uses default procedures to release the storage; otherwise, **ReleaseStgMedium** uses the specified **IUnknown** interface.

See Also

FORMATETC, **IAdviseSink**, **IDataObject**, **IOleCache**, **ReleaseStgMedium**

ACTIVATEFLAGS Quick Info

The **ACTIVATEFLAGS** enumeration value indicates whether an object is activated as a windowless object. It is used in **IOleInPlaceSiteEx::OnInPlaceActivateEx**.

```
typedef enum tagACTIVATEFLAGS
{
    ACTIVATE_WINDOWLESS    = 1
} ACTIVATEFLAGS;
```

Elements

ACTIVATE_WINDOWLESS

If TRUE, indicates that the object is activated in place as a windowless object. In the **IOleInPlaceSiteEx::OnInPlaceActivateEx** method, the container uses this value returned in the *dwFlags* parameter instead of calling the **GetWindow** method in the **IOleInPlaceObjectWindowless** interface to determine if the object is windowless or not.

ADVF Quick Info

The **ADVF** enumeration values are flags used by a container object to specify the requested behavior when setting up an advise sink or a caching connection with an object. These values have different meanings, depending on the type of connection in which they are used, and each interface uses its own subset of the flags.

```
typedef enum tagADVF
{
    ADVF_NODATA            = 1,
    ADVF_PRIMEFIRST        = 4,
    ADVF_ONLYONCE          = 2,
    ADVF_DATAONSTOP        = 64,
    ADVFCACHE_NOHANDLER    = 8,
    ADVFCACHE_FORCEBUILTIN = 16,
    ADVFCACHE_ONSAVE       = 32
} ADVF;
```

Elements

ADVF_NODATA

For data advisory connections (**IDataObject::DAdvise** or **IDataAdviseHolder::Advise**), this flag requests the data object not to send data when it calls **IAdviseSink::OnDataChange**. The recipient of the change notification can later request the data by calling **IDataObject::GetData**. The data object can honor the request by passing TYMED_NULL in the **STGMEDIUM** parameter, or it can provide the data anyway. For example, the data object might have multiple advisory connections, not all of which specified ADVF_NODATA, in which case the object might send the same notification to all connections. Regardless of the container's request, its **IAdviseSink** implementation must check the **STGMEDIUM** parameter because it is responsible for releasing the medium if it is not TYMED_NULL.

For cache connections (**IOleCache::Cache**), this flag requests that the cache not be updated by changes made to the running object. Instead, the container will update the cache by explicitly calling **IOleCache::SetData**. This situation typically occurs when the iconic aspect of an object is being cached.

ADVF_NODATA is not a valid flag for view advisory connections (**IViewObject::SetAdvise**) and it returns E_INVALIDARG.

ADVF_PRIMEFIRST

Requests that the object not wait for the data or view to change before making an initial call to **IAdviseSink::OnDataChange** (for data or view advisory connections) or updating the cache (for cache connections). Used with ADVF_ONLYONCE, this parameter provides an asynchronous **GetData** call.

ADVF_ONLYONCE

Requests that the object make only one change notification or cache update before deleting the connection.

ADVF_ONLYONCE automatically deletes the advisory connection after sending one data or view notification. The advisory sink receives only one **IAdviseSink** call. A nonzero connection identifier is returned if the connection is established, so the caller can use it to delete the connection prior to the first change notification.

For data change notifications, the combination of ADVF_ONLYONCE and ADVF_PRIMEFIRST provides, in effect, an asynchronous **IDataObject::GetData** call.

When used with caching, ADVF_ONLYONCE updates the cache one time only, on receipt of the first OnDataChange notification. After the update is complete, the advisory connection between the object

and the cache is disconnected. The source object for the advisory connection calls the **IAdviseSink::Release** method.

ADVF_DATAONSTOP

For data advisory connections, assures accessibility to data. This flag indicates that when the data object is closing, it should call **IAdviseSink::OnDataChange**, providing data with the call. Typically, this value is used in combination with ADVF_NODATA. Without this value, by the time an **OnDataChange** call without data reaches the sink, the source might have completed its shutdown and the data might not be accessible. Sinks that specify this value should accept data provided in **OnDataChange** if it is being passed, because they may not get another chance to retrieve it.

For cache connections, this flag indicates that the object should update the cache as part of object closure.

ADVF_DATAONSTOP is not a valid flag for view advisory connections.

ADVFCACHE_NOHANDLER

Synonym for ADVFCACHE_FORCEBUILTIN, which is used more often.

ADVFCACHE_FORCEBUILTIN

This value is used by DLL object applications and object handlers that perform the drawing of their objects. ADVFCACHE_FORCEBUILTIN instructs OLE to cache presentation data to ensure that there is a presentation in the cache. This value is not a valid flag for data or view advisory connections. For cache connections, this flag caches data that requires only code shipped with OLE (or the underlying operating system) to be present in order to produce it with **IDataObject::GetData** or **IViewObject::Draw**. By specifying this value, the container can ensure that the data can be retrieved even when the object or handler code is not available.

ADVFCACHE_ONSAVE

For cache connections, this flag updates the cached representation only when the object containing the cache is saved. The cache is also updated when the OLE object transitions from the running state back to the loaded state (because a subsequent save operation would require rerunning the object). This value is not a valid flag for data or view advisory connections.

Remarks

For a data or view advisory connection, the container uses the **ADVF** constants when setting up a connection between an **IAdviseSink** instance and and either an **IDataObject** or **IViewObject** instance. These connections are set up using the **IDataObject::DAdvise**, **IDataAdviseHolder::Advise**, or **IViewObject::SetAdvise** methods.

For a caching connection, the constants are specified in the **IOleCache::Cache** method to indicate the container's requests on how the object should update its cache.

These constants are also used in the *advf* member of the **STATDATA** structure. This structure is used by **IEnumSTATDATA** to describe the enumerated connections, and the *advf* member indicates the flags that were specified when the advisory or cache connection was established. When **STATDATA** is used for an **IOleObject::EnumAdvise** enumerator, the *advf* member is indeterminate.

See Also

IDataAdviseHolder, **IDataObject**, **IEnumSTATDATA**, **IOleCache**, **IViewObject**

BIND_FLAGS Quick Info

The **BIND_FLAGS** enumeration values are used to control aspects of moniker binding operations. The values are used in the **BIND_OPTS** structure. Callers of **IMoniker** methods can specify values from this enumeration, and implementers of **IMoniker** methods can use these values in determining what they should do.

```
typedef enum tagBIND_FLAGS
{
    BIND_MAYBOTHERUSER     = 1,
    BIND_JUSTTESTEXISTENCE = 2,
} BIND_FLAGS;
```

Elements

BIND_MAYBOTHERUSER

If this flag is specified, the moniker implementation can interact with the end user. If not present, the moniker implementation should not interact with the user in any way, such as by asking for a password for a network volume that needs mounting. If prohibited from interacting with the user when it otherwise would, a moniker implementation can use a different algorithm that does not require user interaction, or it can fail with the error MK_MUSTBOTHERUSER.

BIND_JUSTTESTEXISTENCE

If this flag is specified, the caller is not interested in having the operation carried out, but only in learning whether the operation could have been carried out had this flag not been specified. For example, this flag lets the caller indicate only an interest in finding out whether an object actually exists by using this flag in a **IMoniker::BindToObject** call. Moniker implementations can, however, ignore this possible optimization and carry out the operation in full. Callers must be able to deal with both cases.

See Also

BIND_OPTS, **IBindCtx**

BINDSPEED Quick Info

The **BINDSPEED** enumeration values indicate approximately how long the caller will wait to bind to an object. Callers of the **IOleItemContainer::GetObject** method specify values from this enumeration, and implementers of that method use these values as a guideline for how quickly they must complete their operation.

```
typedef enum tagBINDSPEED
{
    BINDSPEED_INDEFINITE    = 1,
    BINDSPEED_MODERATE      = 2,
    BINDSPEED_IMMEDIATE     = 3
} BINDSPEED;
```

Elements

BINDSPEED_INDEFINITE

There is no time limit on the binding operation.

BINDSPEED_MODERATE

The **IOleItemContainer::GetObject** operation must be completed in a moderate amount of time. If this flag is specified, the implementation of **IOleItemContainer::GetObject** should return MK_E_EXCEEEDEDDEADLINE unless the object is one of the following:

- Already in the running state
- A pseudo-object (i.e., an object internal to the item container, such as a cell-range in a spreadsheet or a character-range in a word processor).
- An object supported by an in-process server (so it is always in the running state when it is loaded). In this case, **IOleItemContainer::GetObject** should load the designated object, and, if the **OleIsRunning** function indicates that the object is running, return successfully.

BINDSPEED_IMMEDIATE

The caller will wait only a short time. In this case, **IOleItemContainer::GetObject** should return MK_E_EXCEEEDEDDEADLINE unless the object is already in the running state or is a pseudo-object.

Remarks

The system-supplied item moniker implementation is the primary caller of **IOleItemContainer::GetObject**. The **BINDSPEED** value that it specifies depends on the deadline specified by the caller of the moniker operation.

The deadline is stored in the *dwTickCountDeadline* field of the **BIND_OPTS** structure in the bind context passed to the moniker operation. This value is based on the return value of the **GetTickCount** function. If *dwTickCountDeadline* is zero, indicating no deadline, the item moniker implementation specifies BINDSPEED_INDEFINITE. (This is the default *dwTickCountDeadline* value for a bind context returned by the **CreateBindCtx** function.) If the difference between *dwTickCountDeadline* and the value returned by the **GetTickCount** function is greater than 2500, the item moniker implementation specifies BINDSPEED_MODERATE. If the difference is less than 2500, the item moniker implementation specifies BINDSPEED_IMMEDIATE.

Implementations of **IOleItemContainer::GetObject** can use the **BINDSPEED** value as a shortcut approximation of the binding deadline, or they can use the **IBindCtx** instance parameter to determine the

exact deadline.

See Also

BIND_OPTS, **IBindCtx::GetBindOptions**, **IOleItemContainer::GetObject**

CALLTYPE Quick Info

The **CALLTYPE** enumeration constant specifies the call types used by **IMessageFilter::HandleInComingCall**.

```
typedef enum tagCALLTYPE
{
    CALLTYPE_TOPLEVEL             = 1,
    CALLTYPE_NESTED               = 2,
    CALLTYPE_ASYNC                = 3,
    CALLTYPE_TOPLEVEL_CALLPENDING = 4,
    CALLTYPE_ASYNC_CALLPENDING    = 5
} CALLTYPE;
```

Elements

CALLTYPE_TOPLEVEL

A top-level call has arrived and that the object is not currently waiting for a reply from a previous outgoing call. Calls of this type should always be handled.

CALLTYPE_NESTED

A call has arrived bearing the same logical thread identifier as that of a previous outgoing call for which the object is still awaiting a reply. Calls of this type should always handled.

CALLTYPE_ASYNC

An aysnchronous call has arrived. Calls of this type cannot be rejected. OLE always delivers calls of this type.

CALLTYPE_TOPLEVEL_CALLPENDING

A new top-level call has arrived with a new logical thread identifier and that the object is currently waiting for a reply from a previous outgoing call. Calls of this type may be handled or rejected.

CALLTYPE_ASYNC_CALLPENDING

An asynchronous call has arrived with a new logical thread identifier and that the object is currently waiting for a reply from a previous outgoing call. Calls of this type cannot be rejected.

async call - can NOT be rejected

See Also

IMessageFilter::HandleInComingCall, **IMessageFilter**

CLSCTX Quick Info

Values from the CLSCTX enumeration are used in activation calls to indicate the execution contexts in which an object is to be run. These values are also used in calls to **CoRegisterClassObject** to indicate the set of execution contexts in which a class object is to be made available for requests to construct instances.

```
typedef enum tagCLSCTX
{
    CLSCTX_INPROC_SERVER   = 1,
    CLSCTX_INPROC_HANDLER  = 2,
    CLSCTX_LOCAL_SERVER    = 4
    CLSCTX_REMOTE_SERVER = 16
} CLSCTX;
#define CLSCTX_SERVER     (CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER |
CLSCTX_REMOTE_SERVER)
#define CLSCTX_ALL        (CLSCTX_INPROC_HANDLER | CLSCTX_SERVER)
```

Elements

CLSCTX_INPROC_SERVER

The code that creates and manages objects of this class runs in the same process as the caller of the function specifying the class context.

CLSCTX_INPROC_HANDLER

The code that manages objects of this class is an in-process handler. This is a DLL that runs in the client process and implements client-side structures of this class when instances of the class are accessed remotely.

CLSCTX_LOCAL_SERVER

The EXE code that creates and manages objects of this class is loaded in a separate process space (runs on same machine but in a different process).

CLSCTX_REMOTE_SERVER

A remote machine context. The **LocalServer32** or **LocalService** code that creates and manages objects of this class is run on a different machine.

Defined Terms

CLSCTX_SERVER

Indicates server code, whether in-process, local, or remote. This definition ORs CLSCTX_INPROC_SERVER, CLSCTX_LOCAL_SERVER, and CLSCTX_REMOTE_SERVER.

CLSCTX_ALL

Indicates all class contexts. This definition ORs CLSCTX_INPROC_HANDLER and CLSCTX_SERVER.

Remarks

Values from the CLSCTX enumeration are used in activation calls (**CoCreateInstance**, **CoCreateInstanceEx**, **CoGetClassObject**, etc.) to indicate the preferred execution contexts - in-process, local, or remote - in which an object is to be run. They are also used in calls to **CoRegisterClassObject**

to indicate the set of execution contexts in which a class object is to be made available for requests to construct instances (**IClassFactory::CreateInstance**).

To indicate that more than one context is acceptable, you can string multiple values together with Boolean ORs. The contexts are tried in the order in which they are listed.

The following table shows how other OLE functions and methods that call **CoGetClassObject** use the CLSCTX values:

Function Called	Context Flag Used
OleLoad	CLSCTX_INPROC_HANDLER | CLSCTX_INPROC_SERVER Putting an OLE object into the loaded state requires in-process access; but, it doesn't matter if all of the object's function is presently available.
IRunnableObject::Run	CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER Running an OLE object requires connecting to the full code of the object wherever it is located.
CoUnMarshalInterface	CLSCTX_INPROC_HANDLER Unmarshaling needs the form of the class designed for remote access.
IMoniker::BindToObject, for a file moniker created through a call to **CreateFileMoniker**	In this case, uses CLSCTX_SERVER interally to create the instance after calling GetClassFile to determine the class to be instantiated.

The CLSCTX_REMOTE_SERVER value is added to the CLSCTX enumeration for distributed COM. The CLSCTX_SERVER and CLSCTX_ALL constants are further updated to include the CLSCTX_REMOTE_SERVER value.

Given a set of CLSCTX flags, *dwClsCtx*, the execution context to be used depends on the availability of registered class codes and other parameters according to the following algorithm:

The first part of the processing determines whether CLSCTX_REMOTE SERVER should be specified as follows:

1. If the call specifies either
   a) an explicit **COSERVERINFO** structure indicating a machine different from the current machine, or
   b) there is no explicit COSERVERINFO structure specified, but the specified class is registered with either the **RemoteServerName** or **ActivateAtStorage** named-value.

   then CLSCTX_REMOTE_SERVER is implied and is added to *dwClsCtx*. The second case allows applications written prior to the release of distributed COM to be the configuration of classes for remote activation to be used by client applications available prior to DCOM and the CLSCTX_REMOTE_SERVER flag. The cases in which there would be no explicit **COSERVERINFO** structure are 1) The value is specified as NULL, or 2) It is not one of the function parameters, as would be the case in calls to **CoCreateInstance** or **CoGetClassObject** in existing applications.
2. If the explicit COSERVERINFO parameter indicates the current machine,

CLSCTX_REMOTE_SERVER is removed (if present) from *dwClsCtx*.

The rest of the processing proceeds by looking at the value(s) of *dwClsCtx* in the following sequence.

1. If *dwClsCtx* includes CLSCTX_REMOTE_SERVER and no COSERVERINFO parameter is specified, if the activation request indicates a persistent state from which to initialize the object (with **CoGetInstanceFromFile**, **CoGetInstanceFromIStorage**, or, for a file moniker, in a call to **IMoniker::BindToObject**) and the class has an **ActivateAtStorage** sub-key *or no class registry information whatsoever*, the request to activate and initialize is forwarded to the machine where the persistent state resides. (Refer to the remote activation functions listed in the See Also section for details.)
2. If *dwClsCtx* includes CLSCTX_INPROC_SERVER, the class code in the DLL found under the class's **InprocServer32** key is used if this key exists. The class code will run within the same process as the caller.
3. If dwClsCtx includes CLSCTX_INPROC_HANDLER, the class code in the DLL found under the class's **InprocHandler32** key is used if this key exists. The class code will run within the same process as the caller.
4. If *dwClsCtx* includes CLSCTX_LOCAL_SERVER, the class code in the Win32 service found under the class's **LocalService** key is used if this key exists. If no Win32 service is specified, but an EXE is specified under that same key, the class code associated with that EXE is used. The class code (in either case) will be run in a separate service process on the same machine as the caller.
5. If *dwClsCtx* is set to CLSCTX_REMOTE_SERVER and an additional COSERVERINFO parameter to the function specifies a particular remote machine, a request to activate is forwarded to this remote machine with *dwClsCtx* modified to be CLSCTX_LOCAL_SERVER. The class code will run in its own process on this specific machine, which must be different from that of the caller.
6. Finally, if *dwClsCtx* includes CLSCTX_REMOTE_SERVER and no COSERVERINFO parameter is specified, if a machine name is given under the class's **RemoteServerName** named-value, the request to activate is forwarded to this remote machine with *dwClsCtx* modified to be CLSCTX_LOCAL_SERVER. The class code will run in its own process on this specific machine, which must be different from that of the caller.

See Also

CoCreateInstance, **CoGetClassObject**, **CoRegisterClassObject**, **CoGetInstanceFromFile**, **CoGetInstanceFromIStorage**, **CoCreateInstanceEx**, **COSERVERINFO** structure, Creating an Object through a Class Object, Registering a Running EXE Server

COINIT Quick Info

A set of values from the COINIT enumeration is passed as the *dwCoInit* parameter to **CoInitializeEx**. This value determines the concurrency model used for incoming calls to objects created by this thread. This concurrency model can be either *apartment-threaded* or *multi-threaded*.

The COINIT enumeration is defined as follows:

```
typedef enum tagCOINIT{
  COINIT_APARTMENTTHREADED = 0x2,  // Apartment model
  COINIT_MULTITHREADED     = 0x0,  // OLE calls objects on any thread.
  COINIT_DISABLE_OLE1DDE   = 0x4,  // Don't use DDE for Ole1 support.
  COINIT_SPEED_OVER_MEMORY = 0x8,  // Trade memory for speed.
} COINIT;
```

Members

COINIT_MULTITHREADED

Initializes the thread for *multi-threaded* object concurrency (see Remarks).

COINIT_APARTMENTTHREADED

Initializes the thread for *apartment-threaded* object concurrency (see Remarks).

COINIT_DISABLE_OLE1DDE

Disables DDE for Ole1 support.

COINIT_SPEED_OVER_MEMORY

Trades memory for speed.

Remarks

When a thread is initialized through a call to **CoInitializeEx**, you choose whether to initialize it as apartment-threaded or multi-threaded by designating one of the members of COINIT as its second parameter. This designates how incoming calls to any object created by that thread are handled, that is, the object's concurrency.

Apartment-threading, the default model for earlier versions of Windows NT, while allowing for multiple threads of execution, serializes all incoming calls by requiring that calls to methods of objects created by this thread always run on the same thread - the apartment/thread that created them. In addition, calls can arrive only at message-queue boundaries (i.e., only during a **PeekMessage**, **SendMessage**, **DispatchMessage**, etc.). Because of this serialization, it is not typically necessary to write concurrency control into the code for the object, other than to avoid calls to **PeekMessage** and **SendMessage** during processing that must not be interrupted by other method invocations or calls to other objects in the same apartment/thread.

Multi-threading (also called free-threading) allows calls to methods of objects created by this thread to be run on any thread. There is no serialization of calls - many calls may occur to the same method or to the same object or simultaneously. Multi-threaded object concurrency offers the highest performance and takes the best advantage of multi-processor hardware for cross-thread, cross-process, and cross-machine calling, since calls to objects are not serialized in any way. This means, however, that the code for objects must enforce its own concurrency model, typically through the use of Win32 synchronization primitives, such as *critical sections*, *semaphores*, or *mutexes*. In addition, because the object doesn't control the lifetime of the threads that are accessing it, no thread-specific state may be stored in the object (in Thread-Local-Storage).

See Also

CoInitializeEx, Processes and Threads

DATADIR Quick Info

The **DATADIR** enumeration values specify the direction of the data flow in the *dwDirection* parameter of the **IDataObject::EnumFormatEtc** method. This determines the formats that the resulting enumerator can enumerate.

```
typedef enum tagDATADIR
{
    DATADIR_GET             = 1,
    DATADIR_SET             = 2
} DATADIR;
```

Elements

DATADIR_GET

Requests that **IDataObject::EnumFormatEtc** supply an enumerator for the formats that can be specified in **IDataObject::GetData**.

DATADIR_SET

Requests that **IDataObject::EnumFormatEtc** supply an enumerator for the formats that can be specified in **IDataObject::SetData**.

See Also

IDataObject

DISCARDCACHE Quick Info

The **DISCARDCACHE** enumeration values are used in the **IOleCache2::DiscardCache** method to specify what to do with caches that are to be discarded from memory if their dirty bit has been set. The *dwDiscardOptions* parameter specifies whether or not to save these caches.

Defined in the

```
typedef enum tagDISCARDCACHE
{
    DISCARDCACHE_SAVEIFDIRTY    = 0,
    DISCARDCACHE_NOSAVE         = 1
} DISCARDCACHE;
```

Elements

DISCARDCACHE_SAVEIFDIRTY

The cache is to be saved to disk.

DISCARDCACHE_NOSAVE

The cache can be discarded without saving it.

See Also

IOleCache2::DiscardCache

DROPEFFECT Quick Info

The DoDragDrop function and many of the methods in the IDropSource and IDropTarget interfaces pass information about the effects of a drag-and-drop operation in a **DROPEFFECT** enumeration.

Valid drop-effect values are the result of applying the OR operation to the values contained in the **DROPEFFECT** enumeration:

```
typedef enum tagDROPEFFECT
{
    DROPEFFECT_NONE   = 0,
    DROPEFFECT_COPY   = 1,
    DROPEFFECT_MOVE   = 2,
    DROPEFFECT_LINK   = 4,
    DROPEFFECT_SCROLL = 0x80000000
}DROPEFFECT;
```

These values have the following meaning:

DROPEFFECT name	Value	Description
DROPEFFECT_NONE	0	Drop target cannot accept the data.
DROPEFFECT_COPY	1	Drop results in a copy. The original data is untouched by the drag source.
DROPEFFECT_MOVE	2	Drag source should remove the data.
DROPEFFECT_LINK	4	Drag source should create a link to the original data.
DROPEFFECT_SCROLL	0x80000000	Scrolling is about to start or is currently occurring in the target. This value is used in addition to the other values.

Note Your application should always mask values from the **DROPEFFECT** enumeration to ensure compatibility with future implementations. Presently, only four of the 32-bit positions in a **DROPEFFECT** have meaning. In the future, more interpretations for the bits will be added. Drag sources and drop targets should carefully mask these values appropriately before comparing. They should never compare a **DROPEFFECT** against, say, DROPEFFECT_COPY by:

```
if (dwDropEffect == DROPEFFECT_COPY)...
```

Instead, the application should always mask for the value or values being sought:

```
if (dwDropEffect & DROPEFFECT_COPY) == DROPEFFECT_COPY)...
```

or

```
if (dwDropEffect & DROPEFFECT_COPY)...
```

This allows for the definition of new drop effects, while preserving backwards compatibility with existing code.

See Also

DoDragDrop, IDropSource, IDropTarget

DVASPECT Quick Info

The **DVASPECT** enumeration values specify the desired data or view aspect of the object when drawing or getting data.

```
typedef enum tagDVASPECT
{
    DVASPECT_CONTENT    = 1,
    DVASPECT_THUMBNAIL  = 2,
    DVASPECT_ICON       = 4,
    DVASPECT_DOCPRINT   = 8
} DVASPECT;
```

Elements

DVASPECT_CONTENT

Provides a representation of an object so it can be displayed as an embedded object inside of a container. This value is typically specified for compound document objects. The presentation can be provided for the screen or printer.

DVASPECT_THUMBNAIL

Provides a thumbnail representation of an object so it can be displayed in a browsing tool. The thumbnail is approximately a 120 by 120 pixel, 16-color (recommended) device-independent bitmap potentially wrapped in a metafile.

DVASPECT_ICON

Provides an iconic representation of an object.

DVASPECT_DOCPRINT

Provides a representation of the object on the screen as though it were printed to a printer using the Print command from the File menu. The described data may represent a sequence of pages.

Remarks

Values of this enumeration are used to define the *dwAspect* field of the **FORMATETC** structure. Only one **DVASPECT** value can be used to specify a single presentation aspect in a **FORMATETC** structure. The **FORMATETC** structure is used in many OLE functions and interface methods that require information on data presentation.

See Also

IAdviseSink, **IDataObject**, **IOleObject**, **IViewObject**, **IViewObject2**, **OleDraw**, **FORMATETC**, **OBJECTDESCRIPTOR**

DVASPECT2 Quick Info

The **DVASPECT2** enumeration value is used in **IViewObject::Draw** to specify new drawing aspects used to optimize the drawing process.

```
typedef enum tagDVASPECT2
{
    DVASPECT_OPAQUE         = 16,
    DVASPECT_TRANSPARENT    = 32
} DVASPECT2;
```

Elements

DVASPECT_OPAQUE

Represents the opaque, easy to clip parts of an object. Objects may or may not support this aspect.

DVASPECT_TRANSPARENT

Represents the transparent or irregular parts of on object, typically parts that are expensive or impossible to clip out. Objects may or may not support this aspect.

Remarks

To support drawing optimizations to reduce flicker, an object needs to be able to draw and return information about three separate aspects of itself:

DVASPECT_CONTENT

Same as before. Specifies the entire content of an object. All objects should support this aspect.

DVASPECT_OPAQUE

Represents the opaque, easy to clip parts of an object. Objects may or may not support this aspect.

DVASPECT_TRANSPARENT

Represents the transparent or irregular parts of on object, typically parts that are expensive or impossible to clip out. Objects may or may not support this aspect.

The container can determine which of these drawing aspects an object supports by calling the new method **IViewObjectEx::GetViewStatus**. Individual bits return information about which aspects are supported. If an object does not support the **IViewObjectEx** interface, it is assumed to support only DVASPECT_CONTENT.

Depending on which aspects are supported, the container can ask the object to draw itself during the front to back pass only, the back to front pass only, or both. The various possible cases are:

- Objects supporting only DVASPECT_CONTENT should be drawn during the back to front pass, with all opaque parts of any overlapping object clipped out. Since all objects should support this aspect, a container not concerned about flickering - maybe because it is drawing in an offscreen bitmap - can opt to draw all objects that way and skip the front to back pass.
- Objects supporting DVASPECT_OPAQUE may be asked to draw this aspect during the front to back pass. The container is responsible for clipping out the object's opaque regions (returned by **IViewObjectEx::GetRegion**) before painting any further object behind it.
- Objects supporting DVASPECT_TRANSPARENT may be asked to draw this aspect during the back to front pass. The container is responsible for clipping out opaque parts of overlapping objects before

letting an object draw this aspect.

Even when DVASPECT_OPAQUE and DVASPECT_TRANSPARENT are supported, the container is free to use these aspects or not. In particular, if it is painting in an offscreen bitmap and consequently is unconcerned about flicker, the container may use DVASPECT_CONTENT and a one-pass drawing only. However, in a two-pass drawing, if the container uses DVASPECT_OPAQUE during the front to back pass, then it must use DVASPECT_TRANSPARENT during the back to front pass to complete the rendering of the object.

See Also

IViewObject::Draw

DVASPECTINFOFLAG Quick Info

The **DVASPECTINFOFLAG** enumeration value is used in the **DVASPECTINFO** structure to indicate whether an object can support optimized drawing of itself.

```
typedef enum tagDVASPECTINFOFLAG
{
    DVASPECTINFOFLAG_CANOPTIMIZE    = 1
} DVASPECTINFOFLAG;
```

Elements

DVASPECTINFOFLAG_CANOPTIMIZE

If TRUE, indicates that the object can support optimized rendering of itself. Since most objects on a form share the same font, background color, and border types, leaving these values in the device context allows the next object to use them without having to re-select them. Specifically, the object can leave the font, brush, and pen selected on return from the **IViewObject::Draw** method instead of deselecting these from the device context. The container then must deselect these values at the end of the overall drawing process. The object can also leave other drawing state changes in the device context, such as the background color, the text color, raster operation code, the current point, the line drawing, and the poly fill mode. The object cannot change state values unless other objects are capable of restoring them. For example, the object cannot leave a changed mode, transformation value, selected bitmap, clip region, or metafile.

See Also

DVASPECTINFO

DVEXTENTMODE Quick Info

The **DVEXTENTMODE** enumeration values are used in **IViewObjectEx::GetNaturalExtent**.

```
typedef enum tagDVEXTENTMODE
{
    DVEXTENT_CONTENT                = 0,
    DVEXTENT_INTEGRAL               = 1
} DVEXTENTMODE;
```

Elements

DVEXTENT_CONTENT

Indicates that the container will ask the object how big it wants to be to exactly fit its content, for example, in snap-to-size operations.

DVEXTENT_INTEGRAL

Indicates that the container will provide a proposed size to the object for its use in resizing.

See Also

IViewObjectEx::GetNaturalExtent

EXTCONN Quick Info

The **EXTCONN** enumeration specifies the type of external connection existing on an embedded object. Currently, the only supported type is EXTCONN_STRONG, meaning that the external connection is a link. This E**XTCONN** constant is used in the **IExternalConnection::AddConnection** and **IExternalConnection::ReleaseConnection** methods.

```
typedef enum tagEXTCONN
{
    EXTCONN_STRONG      = 0X0001,
    EXTCONN_WEAK        = 0X0002,
    EXTCONN_CALLABLE    = 0X0004
} EXTCONN;
```

Elements

EXTCONN_STRONG

If this value is specified, the external connection must keep the object alive until all strong external connections are cleared through **IExternalConnection::ReleaseConnection**.

EXTCONN_WEAK

This value is currently not used.

EXTCONN_CALLABLE

This value is currently not used.

GUIDKIND Quick Info

The **GUIDKIND** enumeration values are flags used to specify the kind of information requested from an object in the **IProvideClassInfo2**.

```
typedef enum tagGUIDKIND
{
    GUIDKIND_DEFAULT_SOURCE_DISP_IID  =  1,
    GUIDKIND_DEFAULT_SOURCE_IID       =  2,
    GUIDKIND_DEFAULT_DISP_IID         =  3,
    GUIDKIND_DEFAULT_IID              =  4,
    GUIDKIND_TLBID                    =  5,
    GUIDKIND_CLSID                    =  6
} GUIDKIND;
```

Members

GUIDKIND_DEFAULT_SOURCE_DISP_IID

The interface identifier (IID) of the object's outgoing dispinterface, labeled [source, default]. The outgoing interface in question must be derived from **IDispatch**.

GUIDKIND_DEFAULT_SOURCE_IID

The interface identifier (IID) of the object's outgoing interface, labeled [source, default]. The outgoing interface can be any COM interface.

GUIDKIND_DEFAULT_DISP_IID

The interface identifier (IID) of the object's [default] dispinterface that best represents the object as a whole. This dispinterface must be derived from **IDispatch.**.

GUIDKIND_DEFAULT_IID

The interface identifier (IID) of the object's [default] interface that best represents the object as a whole. This interface can be any COM interface.

GUIDKIND_TLBID

The GUID identifying the object's current type library.

GUIDKIND_CLSID

The object's CLSID.

See Also

IProvideClassInfo2

HITRESULT Quick Info

The **HITRESULT** enumeration values are used in **IViewObjectEx::QueryHitPoint** and **IViewObjectEx::QueryHitRect**.

```
typedef enum tagHITRESULT
{
    HITRESULT_OUTSIDE               = 0,
    HITRESULT_TRANSPARENT           = 1,
    HITRESULT_CLOSE                 = 2,
    HITRESULT_HIT                   = 3
} HITRESULT;
```

Elements

HITRESULT_OUTSIDE

The specified location is outside the object and not close to the object.

HITRESULT_TRANSPARENT

The specified location is within the bounds of the object, but not close to the image. For example, a point in the middle of a transparent circle could be HITRESULT_TRANSPARENT.

HITRESULT_CLOSE

The specified location is inside the object or is outside the object but is close enough to the object to be considered inside. Small, thin or detailed objects may use this value. Even if a point is outside the bounding rectangle of an object it may still be close. This value is needed for hitting small objects.

HITRESULT_HIT

The specified location is within the image of the object

See Also

IViewObjectEx::QueryHitPoint, **IViewObjectEx::QueryHitRect**

KEYMODIFIERS Quick Info

The **KEYMODIFIERS** enumeration values are flags used in calls to **IOleControlSite::TranslateAccelerator** to describe additional keyboard states that can modify the meaning of the keyboard messages that are also passed into **IOleControlSite::TranslateAccelerator**.

```
typedef enum tagKEYMODIFIERS
{
    KEYMOD_SHIFT       = 0x00000000,
    KEYMOD_CONTROL     = 0x00000001,
    KEYMOD_ALT         = 0x00000002
} KEYMODIFIERS;
```

Elements

KEYMOD_SHIFT

The Shift key is currently depressed.

KEYMOD_CONTROL

The Control key is currently depressed.

KEYMOD_ALT

The Alt key is currently depressed.

See Also

IOleControlSite::TranslateAccelerator

LOCKTYPE Quick Info

The **LOCKTYPE** enumeration values indicate the type of locking requested for the specified range of bytes. The values are used in the **ILockBytes::LockRegion** and **IStream::LockRegion** methods.

```
typedef enum tagLOCKTYPE
{
    LOCK_WRITE      = 1,
    LOCK_EXCLUSIVE  = 2,
    LOCK_ONLYONCE   = 4
} LOCKTYPE;
```

Elements

LOCK_WRITE

If this lock is granted, the specified range of bytes can be opened and read any number of times, but writing to the locked range is prohibited except for the owner that was granted this lock.

LOCK_EXCLUSIVE

If this lock is granted, writing to the specified range of bytes is prohibited except for the owner that was granted this lock.

LOCK_ONLYONCE

If this lock is granted, no other LOCK_ONLYONCE lock can be obtained on the range. Usually this lock type is an alias for some other lock type. Thus, specific implementations can have additional behavior associated with this lock type.

MKRREDUCE Quick Info

The **MKRREDUCE** enumeration constants are used to specify how far the moniker should be reduced. They are used in the **IMoniker::Reduce** method.

```
typedef enum tagMKRREDUCE
{
    MKRREDUCE_ONE           = 3<<16,
    MKRREDUCE_TOUSER        = 2<<16,
    MKRREDUCE_THROUGHUSER   = 1<<16,
    MKRREDUCE_ALL           = 0
} MKRREDUCE;
```

Elements

MKRREDUCE_ONE

Performs only one step of reducing the moniker. In general, the caller must have specific knowledge about the particular kind of moniker to take advantage of this option.

MKRREDUCE_TOUSER

Reduces the moniker to a form that the user identifies as a persistent object. If no such point exists, then this option should be treated as MKRREDUCE_ALL.

MKRREDUCE_THROUGHUSER

Reduces the moniker to where any further reduction would reduce it to a form that the user does not identify as a persistent object. Often, this is the same stage as MKRREDUCE_TOUSER.

MKRREDUCE_ALL

Reduces the moniker until it is in its simplest form, that is, reduce it to itself.

See Also

IMoniker::Reduce

MKSYS Quick Info

The **MKSYS** enumeration constants indicate the moniker's class. They are returned from the **IMoniker::IsSystemMoniker** method. **MKSYS** is defined in Objidl.h.

```
typedef enum tagMKSYS
{
    MKSYS_NONE               = 0,
    MKSYS_GENERICCOMPOSITE   = 1,
    MKSYS_FILEMONIKER        = 2,
    MKSYS_ANTIMONIKER        = 3,
    MKSYS_ITEMMONIKER        = 4,
    MKSYS_POINTERMONIKER     = 5,
    MKSYS_CLASSMONIKER       = 7
} MKSYS;
```

Elements

MKSYS_NONE

Indicates a custom moniker implementation.

MKSYS_GENERICCOMPOSITE

Indicates the system's generic composite moniker class.

MKSYS_FILEMONIKER

Indicates the system's file moniker class.

MKSYS_ANTIMONIKER

Indicates the system's anti-moniker class.

MKSYS_ITEMMONIKER

Indicates the system's item moniker class.

MKSYS_POINTERMONIKER

Indicates the system's pointer moniker class.

MKSYS_CLASSMONIKER

Indicates the system's class moniker class.

See Also

IMoniker::IsSystemMoniker

MSHCTX Quick Info

The **MSHCTX** enumeration constants specify the destination context, which is the process in which the unmarshaling is to be done. These flags are used in the **IMarshal** and **IStdMarshalInfo** interfaces and in the **CoMarshalInterface** and **CoGetStandardMarshal** functions.

```
typedef enum tagMSHCTX
{
    MSHCTX_LOCAL              = 0,
    MSHCTX_NOSHAREDMEM        = 1,
    MSHCTX_DIFFERENTMACHINE   = 2,
    MSHCTX_INPROC             = 3
} MSHCTX;
```

Elements

MSHCTX_LOCAL

The unmarshaling process is local and has shared memory access with the marshaling process.

MSHCTX_NOSHAREDMEM

The unmarshaling process does not have shared memory access with the marshaling process.

MSHCTX_DIFFERENTMACHINE

The unmarshaling process is on a different machine. The marshaling code cannot assume that a particular piece of application code is installed on that machine.

MSHCTX_INPROC

The unmarshaling will be done in another apartment in the same process. If your object supports multiple threads, your custom marshaler can pass a direct pointer instead of creating a proxy object.

See Also

CoGetStandardMarshal, **CoMarshalInterface**, **IMarshal**, **IStdMarshalInfo**

MSHLFLAGS Quick Info

The **MSHLFLAGS** enumeration constants determine why the marshaling is to be done. These flags are used in the **IMarshal** interface and the **CoMarshalInterface** and **CoGetStandardMarshal** functions.

```
typedef enum tagMSHLFLAGS
{
    MSHLFLAGS_NORMAL         = 0,
    MSHLFLAGS_TABLESTRONG    = 1,
    MSHLFLAGS_TABLEWEAK      = 2
} MSHLFLAGS;
```

Elements

MSHLFLAGS_NORMAL

The marshaling is occurring because an interface pointer is being passed from one process to another. This is the normal case. The data packet produced by the marshaling process will be unmarshaled in the destination process. The marshaled data packet can be unmarshaled just once, or not at all. If the receiver unmarshals the data packet successfully, the **CoReleaseMarshalData** function is automatically called on the data packet as part of the unmarshaling process. If the receiver does not or cannot unmarshal the data packet, the sender must call the **CoReleaseMarshalData** function on the data packet.

MSHLFLAGS_TABLESTRONG

The marshaling is occurring because the data packet is to be stored in a globally accessible table from which it can be unmarshaled one or more times, or not at all. The presence of the data packet in the table counts as a strong reference to the interface being marshaled, meaning that it is sufficient to keep the object alive. When the data packet is removed from the table, the table implementer must call the **CoReleaseMarshalData** function on the data packet.

MSHLFLAGS_TABLESTRONG is used by the **RegisterDragDrop** function when registering a window as a drop target. This keeps the window registered as a drop target no matter how many times the end user drags across the window. The **RevokeDragDrop** function calls **CoReleaseMarshalData**.

MSHLFLAGS_TABLEWEAK

The marshaling is occurring because the data packet is to be stored in a globally accessible table from which it can be unmarshaled one or more times, or not at all. However, the presence of the data packet in the table acts as a weak reference to the interface being marshaled, meaning that it is not sufficient to keep the object alive. When the data packet is removed from the table, the table implementer must call the **CoReleaseMarshalData** function on the data packet.

MSHLFLAGS_TABLEWEAK is typically used when registering an object in the Running Object Table (ROT). This prevents the object's entry in the ROT from keeping the object alive in the absence of any other connections. See **IRunningObjectTable::Register** for more information.

See Also

CoGetStandardMarshal, **CoMarshalInterface**, **CoReleaseMarshalData**, **IMarshal**

OLECLOSE Quick Info

The **OLECLOSE** enumeration constants are used in the **IOleObject::Close** method to determine whether the object should be saved before closing.

```
typedef enum tagOLECLOSE
{
    OLECLOSE_SAVEIFDIRTY  = 0,
    OLECLOSE_NOSAVE       = 1,
    OLECLOSE_PROMPTSAVE   = 2
} OLECLOSE;
```

Elements

OLECLOSE_SAVEIFDIRTY

The object should be saved if it is dirty.

OLECLOSE_NOSAVE

The object should not be saved, even if it is dirty. This flag is typically used when an object is being deleted.

OLECLOSE_PROMPTSAVE

If the object is dirty, the **IOleObject::Close** implementation should display a dialog box to let the end user determine whether to save the object. However, if the object is in the running state but its user interface is invisible, the end user should not be prompted, and the close should be handled as if OLECLOSE_SAVEIFDIRTY had been specified.

See Also

IOleObject::Close

OLECONTF Quick Info

The **OLECONTF** enumeration indicates the kind of objects to be enumerated by the returned **IEnumUnknown** interface. **OLECONTF** contains a set of bitwise constants used in the **IOleContainer::EnumObjects** method.

```
typedef enum tagOLECONTF
{
    OLECONTF_EMBEDDINGS    = 1;
    OLECONTF_LINKS         = 2;
    OLECONTF_OTHERS        = 4;
    OLECONTF_ONLYUSER      = 8;
    OLECONTF_ONLYIFRUNNING = 16
} OLECONTF;
```

Elements

OLECONTF_EMBEDDINGS

Enumerates the embedded objects in the container.

OLECONTF_LINKS

Enumerates the linked objects in the container.

OLECONTF_OTHERS

Enumerates all objects in the container that are not OLE compound document objects (i.e., objects other than linked or embedded objects). Use this flag to enumerate the container's pseudo-objects.

OLECONTF_ONLYUSER

Enumerates only those objects the user is aware of. For example, hidden named-ranges in Microsoft Excel would not be enumerated using this value.

OLECONTF_ONLYIFRUNNING

Enumerates only those linked or embedded objects that are currently in the running state for this container.

See Also

IEnumUnknown, **IOleContainer::EnumObjects**

OLECREATE Quick Info

Values from the **OLECREATE** enumeration are passed as the *dwFlags* parameter to the **OleCreateXXX** functions to indicate how the creation operation should proceed.

```
typedef enum tagOLECREATE
{
    OLECREATE_LEAVERUNNING = 1,
} OLECREATE;
```

Elements

OLECREATE_LEAVERUNNING

Indicates that the newly created object should be left in the running state upon successful completion of the call.

See Also

OleCreateEx, **OleCreateFromFileEx**, **OleCreateFromDataEx**, **OleCreateLinkEx**, **OleCreateLinkToFileEx**, **OleCreateLinkFromDataEx**

OLEDCFLAGS Quick Info

The **OLEDCFLAGS** enumeration value supplies additional information to the container about the device context that the object has requested. It is used in **IOleInPlaceSiteWindowless::GetDC**.

```
typedef enum tagOLEDCFLAGS
{
    OLEDC_NODRAW      = 0x1,
    OLEDC_PAINTBKGND  = 0x2,
    OLEDC_OFFSCREEN   = 0x4
} OLEDCFLAGS;
```

Elements

OLEDC_NODRAW

Indicates that the object will not use the returned *hDC* for drawing but merely to get information about the display device. In this case, the container can simply pass the window's device context without further processing.

OLEDC_PAINTBKGND

Requests that the container paint the background behind the object before returning the device context. Objects should use this flag when requesting a device context to paint a transparent area.

OLEDC_OFFSCREEN

Indicates that the object prefers to draw into an offscreen device context that should then be copied to the screen. The container can honor this request or not. If this bit is cleared, the container must return an on-screen device context allowing the object to perform direct screen operations such as showing a selection via an XOR operation. An object can specify this value when the drawing operation generates a lot of screen flicker.

See Also

IOleInPlaceSiteWindowless::GetDC

OLEGETMONIKER Quick Info

The **OLEGETMONIKER** enumeration constants indicate the requested behavior of the **IOleObject::GetMoniker** and **IOleClientSite::GetMoniker** methods.

```
typedef enum tagOLEGETMONIKER
{
    OLEGETMONIKER_ONLYIFTHERE = 1,
    OLEGETMONIKER_FORCEASSIGN = 2,
    OLEGETMONIKER_UNASSIGN    = 3,
    OLEGETMONIKER_TEMPFORUSER = 4
} OLEGETMONIKER;
```

Elements

OLEGETMONIKER_ONLYIFTHERE

If a moniker for the object or container does not exist, **GetMoniker** should return E_FAIL and not assign a moniker.

OLEGETMONIKER_FORCEASSIGN

If a moniker for the object or container does not exist, **GetMoniker** should create one.

OLEGETMONIKER_UNASSIGN

IOleClientSite::GetMoniker can release the object's moniker (although it is not required to do so). This constant is not valid in **IOleObject::GetMoniker**.

OLEGETMONIKER_TEMPFORUSER

If a moniker for the object does not exist, **IOleObject::GetMoniker** can create a temporary moniker that can be used for display purposes (**IMoniker::GetDisplayName**) but not for binding. This enables the object server to return a descriptive name for the object without incurring the overhead of creating and maintaining a moniker until a link is actually created.

Remarks

If the OLEGETMONIKER_FORCEASSIGN flag causes a container to create a moniker for the object, the container should notify the object by calling the **IOleObject::SetMoniker** method.

See Also

IMoniker, **IOleClientSite::GetMoniker**, **IOleObject::GetMoniker**

OLELINKBIND Quick Info

The **OLELINKBIND** enumeration constants control binding operations to a link source. They are used in the **IOleLink::BindToSource** method.

```
typedef enum tagOLELINKBIND
{
    OLELINKBIND_EVENIFCLASSDIFF = 1,
} OLELINKBIND;
```

Element

OLELINKBIND_EVENIFCLASSDIFF

The binding operation should proceed even if the current class of the link source is different from the last time the link was bound. For example, the link source could be a Lotus spreadsheet that was converted to an Excel spreadsheet.

See Also

IOleLink::BindToSource

OLEMISC Quick Info

The **OLEMISC** enumeration is a set of bitwise constants that can be combined to describe miscellaneous characteristics of an object or class of objects. A container can call the **IOleObject::GetMiscStatus** method to determine the **OLEMISC** bits set for an object. The values specified in an object server's CLSID\MiscStatus entry in the registration database are based on the **OLEMISC** enumeration. These constants are also used in the *dwStatus* member of the **OBJECTDESCRIPTOR** structure.

```
typedef enum tagOLEMISC // bitwise
{
    OLEMISC_RECOMPOSEONRESIZE            = 1,
    OLEMISC_ONLYICONIC                   = 2,
    OLEMISC_INSERTNOTREPLACE             = 4,
    OLEMISC_STATIC                       = 8,
    OLEMISC_CANTLINKINSIDE               = 16,
    OLEMISC_CANLINKBYOLE1                = 32,
    OLEMISC_ISLINKOBJECT                 = 64,
    OLEMISC_INSIDEOUT                    = 128,
    OLEMISC_ACTIVATEWHENVISIBLE          = 256,
    OLEMISC_RENDERINGISDEVICEINDEPENDENT = 512,
    OLEMISC_INVISIBLEATRUNTIME           = 1024
    OLEMISC_ALWAYSRUN                    = 2048,
    OLEMISC_ACTSLIKEBUTTON               = 4096,
    OLEMISC_ACTSLIKELABEL                = 8192,
    OLEMISC_NOUIACTIVATE                 = 16384,
    OLEMISC_ALIGNABLE                    = 32768,
    OLEMISC_SIMPLEFRAME                  = 65536,
    OLEMISC_SETCLIENTSITEFIRST           = 131072,
    OLEMISC_IMEMODE                      = 262144,
    OLEMISC_IGNOREACTIVATEWHENVISIBLE    = 524288,
    OLEMISC_WANTSTOMENUMERGE             = 1048576,
    OLEMISC_SUPPORTSMULTILEVELUNDO       = 2097152
} OLEMISC;
```

Elements

OLEMISC_RECOMPOSEONRESIZE

When the container resizes the space allocated to displaying one of the object's presentations, the object wants to recompose the presentation. This means that on resize, the object wants to do more than scale its picture. If this bit is set, the container should force the object to the running state and call **IOleObject::SetExtent** with the new size.

OLEMISC_ONLYICONIC

The object has no useful content view other than its icon. From the user's perspective, the Display As Icon checkbox (in the Paste Special dialog box) for this object should always be checked, and should not be uncheckable. Note that such an object should still have a drawable content aspect; it will look the same as its icon view.

OLEMISC_INSERTNOTREPLACE

The object has initialized itself from the data in the container's current selection. Containers should examine this bit after calling **IOleObject::InitFromData** to initialize an object from the current selection. If set, the container should insert the object beside the current selection rather than replacing the current selection. If this bit is not set, the object being inserted replaces the current selection.

OLEMISC_STATIC

This object is a static object, which is an object that contains only a presentation; it contains no native data. See **OleCreateStaticFromData**.

OLEMISC_CANTLINKINSIDE

This object cannot be the link source that when bound to activates (runs) the object. If the object is selected and copied to the clipboard, the object's container can offer a link in a clipboard data transfer that, when bound, must connect to the outside of the object. The user would see the object selected in its container, not open for editing. Rather than doing this, the container can simply refuse to offer a link source when transferring objects with this bit set. Examples of objects that have this bit set include OLE1 objects, static objects, and links.

OLEMISC_CANLINKBYOLE1

This object can be linked to by OLE 1 containers. This bit is used in the *dwStatus* member of the **OBJECTDESCRIPTOR** structure transferred with the Object and Link Source Descriptor formats. An object can be linked to by OLE 1 containers if it is an untitled document, a file, or a selection of data within a file. Embedded objects or pseudo-objects that are contained within an embedded object cannot be linked to by OLE 1 containers (i.e., OLE 1 containers cannot link to link sources that, when bound, require more than one object server to be run.

OLEMISC_ISLINKOBJECT

This object is a link object. This bit is significant to OLE 1 and is set by the OLE 2 link object; object applications have no need to set this bit.

OLEMISC_INSIDEOUT

This object is capable of activating in-place, without requiring installation of menus and toolbars to run. Several such objects can be active concurrently. Some containers, such as forms, may choose to activate such objects automatically.

OLEMISC_ACTIVATEWHENVISIBLE

This bit is set only when OLEMISC_INSIDEOUT is set, and indicates that this object prefers to be activated whenever it is visible. Some containers may always ignore this hint.

OLEMISC_RENDERINGISDEVICEINDEPENDENT

This object does not pay any attention to target devices. Its presention data will be the same in all cases.

OLEMISC_INVISIBLEATRUNTIME

This value is used with controls. It indicates that the control has no run-time user interface, but that it should be visible at design time. For example, a timer control that fires a specific event periodically would not show itself at run time, but it needs a design-time user interface so a form designer can set the event period and other properties.

OLEMISC_ALWAYSRUN

This value is used with controls. It tells the container that this control always wants to be running. As a result, the container should call **OleRun** when loading or creating the object.

OLEMISC_ACTSLIKEBUTTON

This value is used with controls. It indicates that the control is buttonlike in that it understands and obeys the container's DisplayAsDefault ambient property.

OLEMISC_ACTSLIKELABEL

This value is used with controls. It marks the control as a label for whatever control comes after it in the form's ordering. Pressing a mnemonic key for a label control activates the control after it.

OLEMISC_NOUIACTIVATE

This value is used with controls. It indicates that the control has no UI active state, meaning that it requires no in-place tools, no shared menu, and no accelerators. It also means that the control never needs the focus.

OLEMISC_ALIGNABLE

This value is used with controls. It indicates that the control understands how to align itself within its display rectangle, according to alignment properties such as left, center, and right.

OLEMISC_SIMPLEFRAME

This value is used with controls. It indicates that the control is a simple grouping of other controls and does little more than pass Windows messages to the control container managing the form. Controls of this sort require the implementation of **ISimpleFrameSite** on the container's site.

OLEMISC_SETCLIENTSITEFIRST

This value is used with controls. It indicates that the control wants to use **IOleObject::SetClientSite** as its initialization function, even before a call such as **IPersistStreamInit::InitNew** or **IPersistStorage::InitNew**. This allows the control to access a container's ambient properties before loading information from persistent storage. Note that the current implementations of **OleCreate**, **OleCreateFromData**, **OleCreateFromFile**, **OleLoad**, and the default handler do not understand this value. Control containers that wish to honor this value must currently implement their own versions of these functions in order to establish the correct initialization sequence for the control.

OLEMISC_IMEMODE

Obsolete. A control that works with an Input Method Editor (IME) system component can control the state of the IME through the IMEMode property rather than using this value in the OLEMISC enumeration. You can use an IME component to enter information in Asian character sets with a regular keyboard. A Japanese IME, for example, allows you to type a word such as "sushi," on a regular keyboard and when you hit the spacebar, the IME component converts that word to appropriate kanji or proposes possible choices. The OLEMISC_IMEMODE value was previously used to mark a control as capable of controlling an IME mode system component.

OLEMISC_IGNOREACTIVATEWHENVISIBLE

For new ActiveX controls to work in an older container, the control may need to have the OLEMISC_ACTIVATEWHENVISIBLE value set. However, in a newer container that understands and uses **IPointerInactive**, the control does not wish to be in-place activated when it becomes visible. To allow the control to work in both kinds of containers, the control can set this value. Then, the container ignores OLEMISC_ACTIVATEWHENVISIBLE and does not in-place activate the control when it becomes visible.

OLEMISC_WANTSTOMENUMERGE

A control that can merge its menu with its container sets this value.

OLEMISC_SUPPORTSMULTILEVELUNDO

A control that supports multi-level undo sets this value.

See Also

IOleObject::GetMiscStatus, **OBJECTDESCRIPTOR**

OLERENDER Quick Info

The **OLERENDER** enumeration constants are used in the various object creation functions to indicate the type of caching requested for the newly created object.

```
typedef enum tagOLERENDER
{
    OLERENDER_NONE   = 0;
    OLERENDER_DRAW   = 1;
    OLERENDER_FORMAT = 2;
    OLERENDER_ASIS   = 3
} OLERENDER;
```

Elements

OLERENDER_NONE

The client is not requesting any locally cached drawing or data retrieval capabilities in the object. The *pFormatEtc* parameter of the calls is ignored when this value is specified for the *renderopts* parameter.

OLERENDER_DRAW

The client will draw the content of the object on the screen (a NULL target device) using **IViewObject:Draw**. The object itself determines the data formats that need to be cached. With this render option, only the *ptd* and *dwAspect* members of *pFormatEtc* are significant, since the object may cache things differently depending on the parameter values. However, *pFormatEtc* can legally be NULL here, in which case the object is to assume the display target device and the DVASPECT_CONTENT aspect.

OLERENDER_FORMAT

The client will pull one format from the object using **IDataObject::GetData()**. The format of the data to be cached is passed in *pFormatEtc*, which may not in this case be NULL.

OLERENDER_ASIS

The client is not requesting any locally cached drawing or data retrieval capabilities in the object. *pFormatEtc* is ignored for this option. The difference between this and the OLERENDER_FORMAT value is important in such functions as **OleCreateFromData()** and **OleCreateLinkFromData()**.

See Also

OleCreate, **OleCreateFromData**, **OleCreateFromFile**, **OleCreateLink**, **OleCreateLinkFromData**, **OleCreateLinkToFile**, **OleCreateStaticFromData**

OLEUIPASTEFLAG Quick Info

This enumeration is used to indicate the user options that are available to the user when pasting this format, and within which group or list of choices (Paste, Paste Link, etc.) this entry is to be available. OLEUIPASTEFLAG is used by the **OLEUIPASTEENTRY** structure.

```
typedef enum tagOLEUIPASTEFLAG
{
    OLEUIPASTE_ENABLEICON    = 2048,
    OLEUIPASTE_PASTEONLY     = 0,
    OLEUIPASTE_PASTE         = 512,
    OLEUIPASTE_LINKANYTYPE   = 1024,
    OLEUIPASTE_LINKTYPE1     = 1,
    OLEUIPASTE_LINKTYPE2     = 2,
    OLEUIPASTE_LINKTYPE3     = 4,
    OLEUIPASTE_LINKTYPE4     = 8,
    OLEUIPASTE_LINKTYPE5     = 16,
    OLEUIPASTE_LINKTYPE6     = 32,
    OLEUIPASTE_LINKTYPE7     = 64,
    OLEUIPASTE_LINKTYPE8     = 128
} OLEUIPASTEFLAG;
```

Elements

OLEUIPASTE_ENABLEICON

If the container does not specify this flag for the entry in the **OLEUIPASTEENTRY**array passed as input to **OleUIPasteSpecial**, the DisplayAsIcon button will be unchecked and disabled when the user selects the format that corresponds to the entry.

OLEUIPASTE_PASTEONLY

The entry in the OLEUIPASTEENTRY array is valid for pasting only.

OLEUIPASTE_PASTE

The entry in the **OLEUIPASTEENTRY**array is valid for pasting. It may also be valid for linking if any of the following linking flags are specified. If it is valid for linking, then the following flags indicate which link types are acceptable by OR'ing together the appropriate OLEUIPASTE_LINKTYPE<#> values. These values correspond as follows to the array of link types passed to **OleUIPasteSpecial**:

OLEUIPASTE_LINKTYPE1=arrLinkTypes[0]
OLEUIPASTE_LINKTYPE2=arrLinkTypes[1]
OLEUIPASTE_LINKTYPE3=arrLinkTypes[2]
OLEUIPASTE_LINKTYPE4=arrLinkTypes[3]
OLEUIPASTE_LINKTYPE5=arrLinkTypes[4]
OLEUIPASTE_LINKTYPE6=arrLinkTypes[5]
OLEUIPASTE_LINKTYPE7=arrLinkTypes[6]
OLEUIPASTE_LINKTYPE8=arrLinkTypes[7]

where,

UINT arrLinkTypes[8] is an array of registered clipboard formats for linking. A maximum of 8 link types is allowed.

OLEUPDATE Quick Info

The **OLEUPDATE** enumeration constants are used to indicate whether the linked object updates the cached data for the linked object automatically or only when the container calls either the **IOleObject::Update** or **IOleLink::Update** methods. The constants are used in the **IOleLink** interface.

```
typedef enum tagOLEUPDATE
{
    OLEUPDATE_ALWAYS    = 1;
    OLEUPDATE_ONCALL    = 3
} OLEUPDATE;
typedef OLEUPDATE *LPOLEUPDATE;
```

Elements

OLEUPDATE_ALWAYS

Update the link object whenever possible, this option corresponds to the Automatic update option in the Links dialog box.

OLEUPDATE_ONCALL

Update the link object only when **IOleObject::Update** or **IOleLink::Update** is called, this option corresponds to the Manual update option in the Links dialog box.

See Also

IOleLink::SetUpdateOptions, **IOleLink::GetUpdateOptions**

OLEVERBATTRIB Quick Info

The **OLEVERBATTRIB** enumeration constants are used in the **OLEVERB** structure to describe the attributes of a specified verb for an object. Values are used in the enumerator (which supports the **IEnumOLEVERB** interface) that is created by a call to **IOleObject::EnumVerbs**.

```
typedef enum tagOLEVERBATTRIB
{
    OLEVERBATTRIB_NEVERDIRTIES       = 1,
    OLEVERBATTRIB_ONCONTAINERMENU    = 2
} OLEVERBATTRIB;
```

Elements

OLEVERBATTRIB_NEVERDIRTIES

Executing this verb will not cause the object to become dirty and is therefore in need of saving to persistent storage.

OLEVERBATTRIB_ONCONTAINERMENU

Indicates a verb that should appear in the container's menu of verbs for this object. OLEIVERB_HIDE, OLEIVERB_SHOW, and OLEIVERB_OPEN never have this value set.

See Also

IOleObject::EnumVerbs, **IEnumOLEVERB**, **OLEVERB**

OLEWHICHMK Quick Info

The **OLEWHICHMK** enumeration constants indicate which part of an object's moniker is being set or retrieved. These constants are used in the **IOleObject** and **IOleClientSite** interfaces.

```
typedef enum tagOLEWHICHMK
{
    OLEWHICHMK_CONTAINER    = 1,
    OLEWHICHMK_OBJREL       = 2,
    OLEWHICHMK_OBJFULL      = 3
} OLEWHICHMK;
```

Elements

OLEWHICHMK_CONTAINER

The moniker of the object's container. Typically, this is a file moniker. This moniker is not persistently stored inside the object, since the container can be renamed even while the object is not loaded.

OLEWHICHMK_OBJREL

The moniker of the object relative to its container. Typically, this is an item moniker, and it is part of the persistent state of the object. If this moniker is composed on to the end of the container's moniker, the resulting moniker is the full moniker of the object.

OLEWHICHMK_OBJFULL

The full moniker of the object. Binding to this moniker results in a connection to the object. This moniker is not persistently stored inside the object, since the container can be renamed even while the object is not loaded.

See Also

IOleClientSite::GetMoniker, **IOleObject::GetMoniker**, **IOleObject::SetMoniker**

PENDINGMSG Quick Info

The **PENDINGMSG** enumeration constants are return values of **IMessageFilter::MessagePending**.

Defined in the **IMessageFilter** interface (*msgflt.idl*).

```
typedef enum tagPENDINGMSG
{
    PENDINGMSG_CANCELCALL        = 0,
    PENDINGMSG_WAITNOPROCESS     = 1,
    PENDINGMSG_WAITDEFPROCESS    = 2
} PENDINGMSG;
```

Elements

PENDINGMSG_CANCELCALL

Cancel the outgoing call.

PENDINGMSG_WAITNOPROCESS

Wait for the return and don't dispatch the message.

PENDINGMSG_WAITDEFPROCESS

Wait and dispatch the message.

See Also

IMessageFilter::MessagePending

PENDINGTYPE Quick Info

The **PENDINGTYPE** enumeration constants indicate the level of nesting in the **IMessageFilter::MessagePending** method.

```
typedef enum tagPENDINGTYPE
{
    PENDINGTYPE_TOPLEVEL    = 1,
    PENDINGTYPE_NESTED      = 2
} PENDINGTYPE;
```

Elements

PENDINGTYPE_TOPLEVEL

Top-level call.

PENDINGTYPE_NESTED

Nested call.

See Also

IMessageFilter::MessagePending

PICTURE Quick Info

The **PICTURE** enumeration values describe attributes of a picture object as returned through the **IPicture::get_Attributes** method.

```
typedef enum tagPICTURE
{
    PICTURE_SCALABLE       = 0x00000001,
    PICTURE_TRANSPARENT    = 0x00000002
} PICTURE;
```

Elements

PICTURE_SCALABLE

The picture object is scalable, such that it can be redrawn with a different size than was used to create the picture originally. Metafile-based pictures are considered scalable; icon and bitmap pictures, while they can be scaled, do not express this attribute because both involve bitmap stretching instead of true scaling.

PICTURE_TRANSPARENT

The picture object contains an image that has transparent areas, such that drawing the picture will not necessarily fill in all the spaces in the rectangle it occupies. Metafile and icon pictures have this attribute; bitmap pictures do not.

See Also

IPicture::get_Attributes

PICTYPE Quick Info

The **PICTYPE** enumeration values are used to describe the type of a picture object as returned by **IPicture::get_Type,** as well as to describe the type of picture in the **picType** field of the **PICTDESC** structure that is passed to **OleCreatePictureIndirect**.

```
typedef enum tagPICTYPE
{
    PICTYPE_UNINITIALIZED    = -1,
    PICTYPE_NONE             = 0,
    PICTYPE_BITMAP           = 1,
    PICTYPE_METAFILE         = 2,
    PICTYPE_ICON             = 3
    PICTYPE_ENHMETAFILE      = 4
} PICTYPE;
```

Elements

PICTYPE_UNINITIALIZED

The picture object is currently uninitialized. This value is only valid as a return value from **IPicture::get_Type** and is not valid with the **PICTDESC** structure.

PICTYPE_NONE

A new picture object is to be created without an initialized state. This value is valid only in the **PICTDESC** structure.

PICTYPE_BITMAP

The picture type is a bitmap. When this value occurs in the **PICTDESC** structure, it means that the **bmp** field of that structure contains the relevant initialization parameters.

PICTYPE_METAFILE

The picture type is a metafile. When this value occurs in the **PICTDESC** structure, it means that the **wmf** field of that structure contains the relevant initialization parameters.

PICTYPE_ICON

The picture type is an icon. When this value occurs in the **PICTDESC** structure, it means that the **icon** field of that structure contains the relevant initialization parameters.

PICTYPE_ENHMETAFILE

The picture type is a Win32-enhanced metafile. When this value occurs in the **PICTDESC** structure, it means that the **emf** field of that structure contains the relevant initialization parameters.

See Also

IPicture::get_Type, **OleCreatePictureIndirect**, **PICTDESC**

POINTERINACTIVE Quick Info

The **POINTERINACTIVE** enumeration values indicate the activation policy of the object and are used in the **IPointerInactive::GetActivationPolicy** method.

```
typedef enum tagPOINTERINACTIVE
{
    POINTERINACTIVE_ACTIVATEONENTRY      = 1,
    POINTERINACTIVE_DEACTIVATEONLEAVE    = 2,
    POINTERINACTIVE_ACTIVATEONDRAG       = 4
} POINTERINACTIVE;
```

Elements

POINTERINACTIVE_ACTIVATEONENTRY

The object should be in-place activated when the mouse enters it during a mouse move operation.

POINTERINACTIVE_DEACTIVATEONLEAVE

The object should be deactivated when the mouse leaves the object during a mouse move operation.

POINTERINACTIVE_ACTIVATEONDRAG

The object should be in-place activated when the mouse is dragged over it during a drag and drop operation.

Remarks

For more information on using the POINTERINACTIVE_ACTIVATEONENTRY and POINTERINACTIVE_DEACTIVATEONLEAVE values, see the **IPointerInactive::GetActivationPolicy** method.

The POINTERINACTIVE_ACTIVATEONDRAG value can be used to support drag and drop operations on an inactive object. An inactive object has no window to register itself as a potential drop target. Most containers ignore embedded, inactive objects as drop targets because of the overhead associated with activating them.

As an alternative to activating an object when the mouse pointer is over it during a drag and drop operation, the container can first **QueryInterface** to determine if the inactive object supports **IPointerInactive**. Then, if the object does not support **IPointerInactive**, the container can assume that it is not a drop target. If the object does support **IPointerInactive**, the container calls the **IPointerInactive::GetActivationPolicy** method. If the POINTERINACTIVE_ACTIVATEONDRAG value is set, the container activates the object in-place so the object can register its own **IDropTarget** interface.

The container is processing its own **IDropTarget::DragOver** method when all these actions occur. To complete that method, the container returns DROPEFFECT_NONE for the *pdwEffect* parameter. Then, the drag and drop operation continues by calling the container's **IDropTarget::DragLeave** and then calling the object's **IDropTarget::DragEnter**.

Note For windowless OLE objects this mechanism is slightly different. See **IOleInPlaceSiteWindowless** for more information on drag and drop operations for windowless objects.

If the drop occurs on the embedded object, the object is UI-activated and will get UI-deactivated when the

focus changes again. If the drop does not occur on the object, the container should deactivate the object the next time it gets a call to its own **IDropTarget::DragEnter**. It is possible for the drop to occur on a third active object without an intervening call to the container's **IDropTarget::DragEnter**. In this case, the container should try to deactivate the object as soon as it can, for example, when it UI-activates another object.

See Also

IDropTarget, **IPointerInactive::GetActivationPolicy**

PROPSETFLAG Quick Info

The PROPSETFLAG enumeration values define characteristics of a property set. The values are used in the **IPropertySetStorage::Create** method.

```
typedef enum PROPSETFLAG {
    PROPSETFLAG_DEFAULT      = 0,
    PROPSETFLAG_NONSIMPLE    = 1,
    PROPSETFLAG_ANSI         = 2,
    } PROPSETFLAG
```

Members

PROPSETFLAG_NONSIMPLE

If specified, storage-valued and stream-valued properties are permitted in the newly created set. Otherwise, they are not permitted. In the compound file implementation, property sets may be transacted only if PROPSETFLAG_NONSIMPLE is specified.

PROPSETFLAG_ANSI

If specified, all string values in the property set that are not explicitly Unicode (those other than VT_LPWSTR) are stored with the current system ANSI code page (see the Win32 function **GetACP**). Use of this flag is not recommended, as described in the following Remarks section.

If this flag is absent, string values in the new property set are stored in Unicode. The degree of control afforded by this flag is necessary so clients using the property-related interfaces can interoperate well with standard property sets such as the OLE2 summary information, which may exist in the ANSI code page.

Remarks

These values can be set and checked for using bitwise operations, permitting up to four possible combinations: non-simple Unicode, simple Unicode, non-simple ANSI, and simple ANSI.

It is recommended that property sets be created as Unicode, by not setting the PROPSETFLAG_ANSI flag in the *grfFlags* parameter of **IPropertySetStorage::Create**. It is also recommended that you avoid using VT_LPSTR values, and use VT_LPWSTR values instead. When the property set code page is Unicode, VT_LPSTR string values are converted to Unicode when stored, and back to multibyte string values when retrieved. When the code page of the property set is not Unicode, property names, VT_BSTR strings, and non-simple property values are converted to multibyte strings when stored, and converted back to Unicode when retrieved, all using the current system ANSI code page.

See Also

IPropertySetStorage::Create, **IPropertySetStorage::Open**

QACONTAINERFLAGS Quick Info

The **QACONTAINERFLAGS** enumeration value indicates ambient properties supplied by the container. It is used in the *dwAmbientFlags* member of the **QACONTAINER** structure.

```
typedef enum tagQACONTAINERFLAGS
{
    QACONTAINER_SHOWHATCHING       = 0x1,
    QACONTAINER_SHOWGRABHANDLES    = 0x2,
    QACONTAINER_USERMODE           = 0x4,
    QACONTAINER_DISPLAYASDEFAULT   = 0x8,
    QACONTAINER_UIDEAD             = 0x10,
    QACONTAINER_AUTOCLIP           = 0x20,
    QACONTAINER_MESSAGEREFLECT     = 0x40,
    QACONTAINER_SUPPORTSMNEMONICS  = 0x80
} QACONTAINERFLAGS;
```

Elements

QACONTAINER_SHOWHATCHING

Specifies the ShowHatching ambient property, which has a standard ambient DISPID of -712.

QACONTAINER_SHOWGRABHANDLES

Specifies the ShowGrabHandles ambient property, which has a standard ambient DISPID of -711.

QACONTAINER_USERMODE

Specifies the UserMode ambient property, which has a standard ambient DISPID of -709.

QACONTAINER_DISPLAYASDEFAULT

Specifies the DisplayAsDefault ambient property, which has a standard ambient DISPID of -713.

QACONTAINER_UIDEAD

Specifies the UIDead ambient property, which has a standard ambient DISPID of -710.

QACONTAINER_AUTOCLIP

Specifies the AutoClip ambient property, which has a standard ambient DISPID of -715.

QACONTAINER_MESSAGEREFLECT

Specifies the MessageReflect ambient property, which has a standard ambient DISPID of -706.

QACONTAINER_SUPPORTSMNEMONICS

Specifies the SupportsMnemonics ambient property, which has a standard ambient DISPID of -714.

Remarks

See the ActiveX Controls chapter in the OLE Programmer's Guide for further information on standard ambient properties.

See Also

QACONTAINER

REGCLS Quick Info

The **REGCLS** enumeration defines flags used in **CoRegisterClassObject** to control the type of connections to the class object. It is defined as follows:

```
typedef enum tagREGCLS
{
    REGCLS_SINGLEUSE          = 0,
    REGCLS_MULTIPLEUSE        = 1,
    REGCLS_MULTI_SEPARATE     = 2,
} REGCLS;
```

Elements

REGCLS_SINGLEUSE

Once an application has connected to the class object with **CoGetClassObject**, the class object is removed from public view so that no other applications can connect to it. This flag is commonly used for single document interface (SDI) applications. Specifying this flag does not affect the responsibility of the object application to call **CoRevokeClassObject**; it must always call **CoRevokeClassObject** when it is finished with an object class.

REGCLS_MULTIPLEUSE

Multiple applications can connect to the class object through calls to **CoGetClassObject**.

REGCLS_MULTI_SEPARATE

Similar to REGCLS_MULTIPLEUSE, except that REGCLS_MULTI_SEPARATE does not automatically register the class object as CLSCTX_INPROC_SERVER for a local server. Instead, it provides separate control over each context. When a class is registered this way, if that server tries to bind to an object with its own class identifier, it will start another copy of the server.

Remarks

In **CoRegisterClassObject**, members of both the **REGCLS** and the **CLSCTX** enumerations, taken together, determine how the class object is registered.

The following table summarizes the allowable flag combinations and the object registrations affected by the combinations.

	REGCLS_ SINGLEUSE	**REGCLS_ MULTIPLEUSE**	**REGCLS_ MULTI_ SEPARATE**	**Other**
CLSCTX_ INPROC_ SERVER	Error	Inproc	Inproc	Error
CLSCTX_ LOCAL_ SERVER	Local	Inproc/local	Local	Error
Both of the above	Error	Inproc/local	Inproc/local	Error
Other	Error	Error	Error	Error

REGCLS_MULTIPLEUSE in combination with CLSCTX_LOCAL_SERVER automatically registers the class object as an in-process server (CLSCTX_INPROC_SERVER). In contrast, registering a class object as a local server and specifying REGCLS_MULTIPLE_SEPARATE does not register the class object as an in-process server (registers the object with the CLSCTX_LOCAL_SERVER flag, but does not automatically add the CLSCTX_INPROC_SERVER flag, as is the case when you specify the REGCLS_MULTIPLEUSE flag. This distinction is important in applications that are both OLE containers and OLE embeddings, allowing a container/server to be inserted into itself.

In general, the following two registrations have the same effect -- they register class objects as both multiple-use and as in-process servers:

```
CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE

(CLSCTX_INPROC_SERVER|CLSCTX_LOCAL_SERVER), REGCLS_MULTI_SEPARATE
```

The following registers the class object only as a multiple-use local server:

```
CLSCTX_LOCAL_SERVER, REGCLS_MULTI_SEPARATE
```

See Also

CoGetClassObject, **CoRegisterClassObject**, **CoRevokeClassObject**, **DllGetClassObject**

RPC_C_AUTHN_*xxx* Quick Info

These values, along with the RPC_C_AUTHZ_*xxx* values, are assigned to the SOLE_AUTHENICATION_SERVICE structure, which is retrieved by the **CoQueryAuthenticationServices** function, and passed in to the **CoInitializeSecurity** function.

Note Only RPC_C_AUTHN_WINNT is supported in NT 4.0. The others may be supported with software purchased from other companies.

Values

RPC_C_AUTHN_NONE

No authentication.

RPC_C_AUTHN_DCE_PRIVATE

DCE private key authentication.

RPC_C_AUTHN_DCE_PUBLIC

DCE public key authentication.

RPC_C_AUTHN_DEC_PUBLIC

DEC public key authentication (reserved for future use).

RPC_C_AUTHN_WINNT

NT LM SSP (NT Security Service).

RPC_C_AUTHN_DEFAULT

The system default authentication service. Windows NT 4.0 defaults to DCE private key authentication (RPC_C_AUTHN_DCE_PRIVATE).

See Also

CoInitializeSecurity, **CoQueryAuthenticationServices**

RPC_C_AUTHN_LEVEL_xxx Quick Info

Used in the security functions and interfaces to specify the authentication level.

Values

RPC_C_AUTHN_LEVEL_NONE

Performs no authentication.

RPC_C_AUTHN_LEVEL_CONNECT

Authenticates only when the client establishes a relationship with the server. Datagram transports always use RPC_AUTHN_LEVEL_PKT instead.

RPC_C_AUTHN_LEVEL_CALL

Authenticates only at the beginning of each remote procedure call when the server receives the request. Datagram transports use RPC_C_AUTHN_LEVEL_PKT instead.

RPC_C_AUTHN_LEVEL_PKT

Authenticates that all data received is from the expected client.

RPC_C_AUTHN_LEVEL_PKT_INTEGRITY

Authenticates and verifies that none of the data transferred between client and server has been modified.

RPC_C_AUTHN_LEVEL_PKT_PRIVACY

Authenticates all previous levels and encrypts the argument value of each remote procedure call.

Windows95: While Windows95 can make calls at any level, it can only receive calls at NONE or CONNECT. This applies to RPC today, although Windows95 does not support distributed COM at this time.

See Also

IClientSecurity, **IServerSecurity**

RPC_C_AUTHZ_*xxx* Quick Info

These values define what the server authorizes, and are used by methods of the **IClientSecurity** interface. Along with the RPC_C_AUTHN_*xxx* values, these are the values assigned to the **SOLE_AUTHENTICATION_SERVICE** structure, which is retrieved by the **CoQueryAuthenticationServices** function, and passed in to the **CoInitializeSecurity** function.

Values

RPC_C_AUTHZ_NONE

Server performs no authorization.

RPC_C_AUTHZ_NAME

Server performs authorization based on the client's principal name.

RPC_C_AUTHZ_DCE

Server performs authorization checking using the client's DCE privilege attribute certificate (PAC) information, which is sent to the server with each remote procedure call made using the binding handle. Generally, access is checked against DCE access control lists (ACLs).

See Also

SOLE_AUTHENTICATION_SERVICE

RPC_C_IMP_LEVEL_*xxx* Quick Info

Used in the security functions and interfaces to specify the authentication level.

Values

RPC_C_IMP_LEVEL_ANONYMOUS

(Not supported in this release.) The client is anonymous to the server. The server process cannot obtain identification information about the client and it cannot impersonate the client.

RPC_C_IMP_LEVEL_IDENTIFY

The server can obtain the client's identity. The server can impersonate the client for ACL checking, but cannot access system objects as the client. This information is obtained when the connection is established, not on every call.

Note **GetUserName** will fail while impersonating at identify level. The workaround is to impersonate, **OpenThreadToken**, revert, call GetTokenInformation, and finally, call **LookupAccountSid**.

RPC_C_IMP_LEVEL_IMPERSONATE

The server process can impersonate the client's security context while acting on behalf of the client. This information is obtained when the connection is established, not on every call.

RPC_C_IMP_LEVEL_DELEGATE

(Not supported in this release.) The server process can impersonate the client's security context while acting on behalf of the client. The server process can also make outgoing calls to other servers while acting on behalf of the client. This information is obtained when the connection is established, not on every call.

Comments

Only the RPC_C_IMP_LEVEL_IDENTIFY and RPC_C_IMP_LEVEL_IMPERSONATE levels are supported in NT 4.0.

See Also

CoInitializeSecurity

SERVERCALL Quick Info

The **SERVERCALL** enumeration constants indicate the status of server call – returned by **IMessageFilter::HandleInComingCall** and passed to **IMessageFilter::RetryRejectedCall**.

Defined in the **IMessageFilter** interface (*msgflt.idl*).

```
typedef enum tagSERVERCALL
{
    SERVERCALL_ISHANDLED     = 0,
    SERVERCALL_REJECTED      = 1,
    SERVERCALL_RETRYLATER    = 2
} SERVERCALL;
```

Elements

SERVERCALL_ISHANDLED

The object may be able to process the call.

SERVERCALL_REJECTED

The object cannot handle the call due to an unforeseen problem, such as network unavailability.

SERVERCALL_RETRYLATER

The object cannot handle the call at this time. For example, an application might return this value when it is in a user-controlled modal state.

See Also

IMessageFilter::HandleInComingCall, **IMessageFilter::RetryRejectedCall**

STATFLAG Quick Info

The **STATFLAG** enumeration values indicate whether the method should try to return a name in the *pwcsName* member of the **STATSTG** structure. The values are used in the **ILockBytes::Stat**, **IStorage::Stat**, and **IStream::Stat** methods to save memory when the *pwcsName* member is not needed.

Defined in the **IOLETypes** pseudo-interface (*oletyp.idl*).

```
typedef enum tagSTATFLAG
{
    STATFLAG_DEFAULT    = 0,
    STATFLAG_NONAME     = 1
} STATFLAG;
```

Elements

STATFLAG_DEFAULT

Requests that the statistics include the *pwcsName* member of the **STATSTG** structure.

STATFLAG_NONAME

Requests that the statistics not include the *pwcsName* member of the **STATSTG** structure. If the name is omitted, there is no need for the **Stat** methods to allocate and free memory for the string value for the name and the method can save an **Alloc** and **Free** operation.

See Also

ILockBytes::Stat, **IStorage::Stat**, **IStream::Stat**

STATSTATE

The **STATSTATE** enumeration values indicate state information about the storage object and are used as a mask. The values are used in the **IStorage::SetStateBits** method.

```
typedef enum tagSTATSTATE
{
    STATSTATE_DOC            = 1,
    STATSTATE_CONVERT        = 2,
    STATSTATE_FILESTGSAME    = 4
} STATSTATE;
```

Elements

STATSTATE_DOC

The storage object is a document file. This bit is set on the root storage object as part of a normal File/Save sequence. With nested storage objects, the application manages the storage objects and sets or clears this bit as appropriate. If the nested object is an embedded object, this bit can be ignored. It is cleared in a newly created storage object. However, some applications might use this bit to enable editing an embedded storage object without first copying the object to the file system. For example, a mail application might set this bit for attachments so the attachments can be edited without copying them first to a file.

STATSTATE_CONVERT

A convert operation was done on this storage object while it was in a passive state.

STATSTATE_FILESTGSAME

The embedded object and document representations for the storage object are the same. Thus, the storage object can be saved in a document file simply by copying the storage object bits.

See Also

IStorage::SetStateBits

STGC Quick Info

The **STGC** enumeration constants specify the conditions for performing the commit operation in the **IStorage::Commit** and **IStream::Commit** methods.

Defined in the **IOLETypes** pseudo-interface (*oletyp.idl*).

```
typedef enum tagSTGC
{
    STGC_DEFAULT                                = 0,
    STGC_OVERWRITE                              = 1,
    STGC_ONLYIFCURRENT                          = 2,
    STGC_DANGEROUSLYCOMMITMERELYTODISKCACHE     = 4
} STGC;
```

Elements

STGC_DEFAULT

None of the other values apply. You can specify this condition or some combination of the other three. You would use this value mainly to make your code more readable.

STGC_OVERWRITE

The commit operation can overwrite existing data to reduce overall space requirements. This value is not recommended for typical usage because it is not as robust as the default case. In this case, it is possible for the commit to fail after the old data is overwritten but before the new data is completely committed. Then, neither the old version nor the new version of the storage object will be intact.

You can use this value in cases where:

- the user has indicated a willingness to risk losing the data
- the low memory save sequence will be used to safely save the storage object to a smaller file
- a previous commit returned STG_E_MEDIUMFULL but overwriting the existing data would provide enough space to commit changes to the storage object

Note that the commit operation checks for adequate space before any overwriting occurs. Thus, even with this value specified, if the commit operation fails due to space requirements, the old data will remain safe. The case where data loss can occur is when the commit operation fails due to some reason other than lack of space and the STGC_OVERWRITE value was specified.

STGC_ONLYIFCURRENT

Prevents multiple users of a storage object from overwriting one another's changes. The commit operation occurs only if there have been no changes to the saved storage object since the user most recently opened the storage object. Thus, the saved version of the storage object is the same version that the user has been editing. If other users have changed the storage object, the commit operation fails and returns the STG_E_NOTCURRENT value. You can override this behavior by calling the **Commit** method again using the STGC_DEFAULT value.

STGC_DANGEROUSLYCOMMITMERELYTODISKCACHE

Commits the changes to a write-behind disk cache, but does not save the cache to the disk. In a write-behind disk cache, the operation that writes to disk actually writes to a disk cache, thus increasing performance. The cache is eventually written to the disk, but usually not until after the write operation has already returned. The performance increase comes at the expense of an increased risk of losing data if a problem occurs before the cache is saved and the data in the cache is lost.

If you do not specify this value, then committing changes to root-level storage objects is robust even if a disk cache is used. The two-phase commit process ensures that data is stored on the disk and not just to the disk cache.

Remarks

You can specify STGC_DEFAULT or some combination of the other three values. Typically, you would use STGC_ONLYIFCURRENT to protect the storage object in cases where more than one user can edit the object simultaneously.

See Also

IPropertyStorage, **IStorage**, **IStream**

STGFMT Quick Info

The **STGFMT** enumeration values indicate the format of a storage object and are used in the **STATSTG** structure and in the **StgCreateDocFile** and **StgIsStorageFile** functions.

```
typedef enum tagSTGFMT
{
    STGFMT_DOCUMENT     = 0,
    STGFMT_DIRECTORY    = 1,
    STGFMT_CATALOG      = 2,
    STGFMT_FILE         = 3
} STGFMT;
```

Elements

STGFMT_DOCUMENT

Indicates a document format.

STGFMT_DIRECTORY

Indicates a directory format.

STGFMT_CATALOG

Indicates a catalog format.

STGFMT_FILE

Indicates a file format.

See Also

STATSTG, **StgCreateDocfile**, **StgIsStorageFile**

STGM Quick Info

The **STGM** enumeration values are used in the storage and stream interfaces to indicate the conditions for creating and deleting the object and access modes for the object.

The **STGM** values are used in the **IStorage** and **IStream** interfaces, and in the **StgCreateDocfile** and **StgCreateDocfileOnILockBytes** functions to indicate the conditions for creating and deleting the object and access modes for the object.

STGM values are as follows:

STGM_DIRECT	0x00000000L
STGM_TRANSACTED	0x00010000L
STGM_SIMPLE	0x08000000L
STGM_READ	0x00000000L
STGM_WRITE	0x00000001L
STGM_READWRITE	0x00000002L
STGM_SHARE_DENY_NONE	0x00000040L
STGM_SHARE_DENY_READ	0x00000030L
STGM_SHARE_DENY_WRITE	0x00000020L
STGM_SHARE_EXCLUSIVE	0x00000010L
STGM_PRIORITY	0x00040000L
STGM_DELETEONRELEASE	0x04000000L
STGM_CREATE	0x00001000L
STGM_CONVERT	0x00020000L
STGM_FAILIFTHERE	0x00000000L
STGM_NOSCRATCH	0x00100000L

Elements

STGM_DIRECT. STGM_TRANSACTED, STGM_SIMPLE group:

STGM_DIRECT

In direct mode, each change to a storage element is written as it occurs. This is the default.

STGM_TRANSACTED

In transacted mode, changes are buffered and are written only if an explicit commit operation is called. The changes can be ignored by calling the **Revert** method in the **IStream** or **IStorage** interfaces. The OLE compound file implementation does not support transacted streams, which means that streams can be opened only in direct mode, and you cannot revert changes to them. Transacted storages are, however, supported.

STGM_SIMPLE

STGM_SIMPLE is a mode that provides a much faster implementation of a compound file in a limited, but frequently used case. It is described in detail in the following Remarks section.

STGM_READ, STGM_WRITE, STGM_READWRITE group:

STGM_READ

For stream objects, STGM_READ allows you to call the **IStream::Read** method. For storage objects,

you can enumerate the storage elements and open them for reading.

STGM_WRITE

STGM_WRITE lets you save changes to the object.

STGM_READWRITE

STGM_READWRITE is the combination of STGM_READ and STGM_WRITE.

STGM_SHARE_* group:

STGM_SHARE_DENY_NONE

Specifies that subsequent openings of the object are not denied read or write access.

STGM_SHARE_DENY_READ

Prevents others from subsequently opening the object in STGM_READ mode. It is typically used on a root storage object.

STGM_SHARE_DENY_WRITE

Prevents others from subsequently opening the object in STGM_WRITE mode. This value is typically used to prevent unnecessary copies made of an object opened by multiple users. If this value is not specified, a snapshot is made, independent of whether there are subsequent opens or not. Thus, you can improve performance by specifying this value.

STGM_SHARE_EXCLUSIVE

The combination of STGM_SHARE_DENY_READ and STGM_SHARE_DENY_WRITE.

STGM_PRIORITY

STGM_PRIORITY

Opens the storage object with exclusive access to the most recently committed version. Thus, other users cannot commit changes to the object while you have it open in priority mode. You gain performance benefits for copy operations, but you prevent others from committing changes. So, you should limit the time you keep objects open in priority mode. You must specify STGM_DIRECT and STGM_READ with priority mode.

STGM_DELETEONRELEASE

STGM_DELETEONRELEASE

Indicates that the underlying file is to be automatically destroyed when the root storage object is released. This capability is most useful for creating temporary files.

STGM_CREATE, STGM_CONVERT, STGM_FAILIFTHERE Group

STGM_CREATE

Indicates that an existing storage object or stream should be removed before the new one replaces it. A new object is created when this flag is specified, only if the existing object has been successfully removed.

This flag is used in three situations:

- when you are trying to create a storage object on disk but a file of that name already exists
- when you are trying to create a stream inside a storage object but a stream with the specified

name already exists

- when you are creating a byte array object but one with the specified name already exists

STGM_CONVERT

Creates the new object while preserving existing data in a stream named **CONTENTS**. In the case of a storage object or a byte array, the old data is flattened to a stream regardless of whether the existing file or byte array currently contains a layered storage object.

STGM_FAILIFTHERE

Causes the create operation to fail if an existing object with the specified name exists. In this case, STG_E_FILEALREADYEXISTS is returned. STGM_FAILIFTHERE applies to both storage objects and streams.

STGM_NOSCRATCH

Windows95 only: In transacted mode, a scratch file is usually used to save until the commit operation. Specifying STGM_NOSCRATCH permits the unused portion of the original file to be used as scratch space. This does not affect the data in the original file, and is a much more efficient use of memory.

Remarks

You can combine these flags but you can only choose one flag from each group of related flags. Groups are indicated under the headings in the previous section.

The STGM_SIMPLE flag is applicable only when combined with:

```
STGM_CREATE | STGM_READWRITE | STGM_SHARE_EXCLUSIVE
```

Note that direct mode is implied by the absence of STGM_TRANSACTED.

This mode is useful for applications that perform complete save operations. It has the following constraints:

1. There is no support for substorages.
2. Access to streams follows a linear pattern. Once a stream is released, that stream cannot be opened for read/write operations again. The **IStorage::OpenStream** method is not supported in this implementation.
3. The storage and stream objects cannot be marshaled.
4. Each stream is at least 4096 bytes in length. If fewer than 4096 bytes are written into a stream by the time the stream is released, the stream will be extended to contain 4096 bytes.
5. In this compound file implementation, only a subset of the methods of **IStorage** and **IStream** are available.

Specifically, in simple mode, supported **IStorage** methods are **QueryInterface**, **AddRef**, **Release**, **CreateStream**, **Commit**, and **SetClass**. In addition, **SetElementTimes** is supported with a NULL name, allowing applications to set times on a root storage in simple mode.

Supported **IStream** methods are **QueryInterface**, **AddRef**, **Release**, **Write**, **Seek**, **SetSize**, and **Read**.

All the other methods of **IStorage** and **IStream** return STG_E_INVALIDFUNCTION.

See Also

StgCreateDocfile, **IStream::Read**, **IStorage**, **StgCreateDocfileOnILockBytes**, **StgOpenStorage**, **StgOpenStorageOnILockBytes**

STGMOVE Quick Info

The **STGMOVE** enumeration values indicate whether a storage element is to be moved or copied. They are used in the **IStorage::MoveElementTo** method.

```
typedef enum tagSTGMOVE
{
    STGMOVE_MOVE    = 0,
    STGMOVE_COPY    = 1
} STGMOVE;
```

Elements

STGMOVE_MOVE

Indicates the method should move the data from the source to the destination.

STGMOVE_COPY

Indicates the method should copy the data from the source to the destination. A copy is the same as a move except the source element is not removed after copying the element to the destination. Copying an element on top of itself is undefined.

See Also

IStorage::MoveElementTo

STGTY

Quick Info

The **STGTY** enumeration values are used in the *type* member of the **STATSTG** structure to indicate the type of the storage element. A storage element is a storage object, a stream object, or a byte array object (LOCKBYTES).

```
typedef enum tagSTGTY
{
    STGTY_STORAGE      = 1,
    STGTY_STREAM       = 2,
    STGTY_LOCKBYTES    = 3,
    STGTY_PROPERTY     = 4
} STGTY;
```

Elements

STGTY_STORAGE

Indicates that the storage element is a storage object.

STGTY_STREAM

Indicates that the storage element is a stream object.

STGTY_LOCKBYTES

Indicates that the storage element is a byte array object.

STGTY_PROPERTY

Indicates that the storage element is a property storage object.

See Also

IStream, **STATSTG**

STREAM_SEEK Quick Info

The **STREAM_SEEK** enumeration values specify the origin from which to calculate the new seek pointer location. They are used for the *dworigin* parameter in the **IStream::Seek** method. The new seek position is calculated using this value and the *dlibMove* parameter.

```
typedef enum tagSTREAM_SEEK
{
    STREAM_SEEK_SET    = 0,
    STREAM_SEEK_CUR    = 1,
    STREAM_SEEK_END    = 2
} STREAM_SEEK;
```

Elements

STREAM_SEEK_SET

The new seek pointer is an offset relative to the beginning of the stream. In this case, the *dlibMove* parameter is the new seek position relative to the beginning of the stream.

STREAM_SEEK_CUR

The new seek pointer is an offset relative to the current seek pointer location. In this case, the *dlibMove* parameter is the signed displacement from the current seek position.

STREAM_SEEK_END

The new seek pointer is an offset relative to the end of the stream. In this case, the *dlibMove* parameter is the new seek position relative to the end of the stream.

See Also

IStream::Seek

TYMED Quick Info

The **TYMED** enumeration values indicate the type of storage medium being used in a data transfer. They are used in the **STGMEDIUM** or **FORMATETC** structures.

```
typedef [transmit_as(long)] enum tagTYMED
{
    TYMED_HGLOBAL     = 1;
    TYMED_FILE        = 2;
    TYMED_ISTREAM     = 4;
    TYMED_ISTORAGE    = 8;
    TYMED_GDI         = 16;
    TYMED_MFPICT      = 32;
    TYMED_ENHMF       = 64;
    TYMED_NULL        = 0
} TYMED;
```

Elements

TYMED_HGLOBAL

The storage medium is a global memory handle (HGLOBAL). Allocate the global handle with the GMEM_SHARE flag. If the **STGMEDIUM** *punkForRelease* member is NULL, the destination process should use **GlobalFree** to release the memory.

TYMED_FILE

The storage medium is a disk file identified by a path. If the **STGMEDIUM** *punkForRelease* member is NULL, the destination process should use **OpenFile** to delete the file.

TYMED_ISTREAM

The storage medium is a stream object identified by an **IStream** pointer. Use **IStream::Read** to read the data. If the **STGMEDIUM** *punkForRelease* member is NULL, the destination process should use **IStream::Release** to release the stream component.

TYMED_ISTORAGE

The storage medium is a storage component identified by an **IStorage** pointer. The data is in the streams and storages contained by this **IStorage** instance. If the **STGMEDIUM** *punkForRelease* member is NULL, the destination process should use **IStorage::Release** to release the storage component.

TYMED_GDI

The storage medium is a GDI component (HBITMAP). If the **STGMEDIUM** *punkForRelease* member is NULL, the destination process should use **DeleteObject** to delete the bitmap.

TYMED_MFPICT

The storage medium is a metafile (HMETAFILE). Use the Windows or WIN32 functions to access the metafile's data. If the **STGMEDIUM** *punkForRelease* member is NULL, the destination process should use **DeleteMetaFile** to delete the bitmap.

TYMED_ENHMF

The storage medium is an enhanced metafile. If the **STGMEDIUM** *punkForRelease* member is NULL, the destination process should use **DeleteEnhMetaFile** to delete the bitmap.

TYMED_NULL

No data is being passed.

Remarks

During data transfer operations, a storage medium is specified. This medium must be released after the data transfer operation. The provider of the medium indicates its choice of ownership scenarios in the value it provides in the **STGMEDIUM** structure. A NULL value for the **IUNKNOWN** field indicates that the receiving body of code owns and can free the medium. A non-NULL pointer specifies that **ReleaseStgMedium** can always be called to free the medium.

See Also

FORMATETC, **IAdviseSink**, **IDataObject**, **IOleCache**, **ReleaseStgMedium**, **STGMEDIUM**

UASFLAGS Quick Info

The **UASFLAGS** enumeration value supplies information about the parent undo unit. It is used in **IOleParentUndoUnit::GetParentState**.

```
typedef enum tagUASFLAGS
{
    UAS_NORMAL          = 0,
    UAS_BLOCKED         = 0x1,
    UAS_NOPARENTENABLE  = 0x2,
    UAS_MASK            = 0x3,
} UASFLAGS;
```

Elements

UAS_NORMAL

The currently open parent undo unit is in a normal, unblocked state and can accept any new units added through calls to its **Open** or **Add** methods.

UAS_BLOCKED

The currently open undo unit is blocked and will reject any undo units added through calls to its **Open** or **Add** methods. The caller need not create any new units since they will just be rejected.

UAS_NOPARENTENABLE

The currently open undo unit will accept new units, but the caller should act like there is no currently open unit. This means that if the new unit being created requires a parent, then this parent does not satisfy that requirement and the undo stack should be cleared.

UAS_MASK

When checking for a normal state, use this value to mask unused bits in the *pdwState* parameter to the **GetParentState** method for future compatibility. For example:

```
fNormal = ((pdwState & UAS_MASK) == UAS_NORMAL)
```

See Also

IOleParentUndoUnit::GetParentState

USERCLASSTYPE Quick Info

The **USERCLASSTYPE** enumeration constants indicate the different variants of the display name associated with a class of objects. They are used in the **IOleObject::GetUserType** method and the **OleRegGetUserType** function.

```
typedef enum tagUSERCLASSTYPE
{
    USERCLASSTYPE_FULL       = 1,
    USERCLASSTYPE_SHORT      = 2,
    USERCLASSTYPE_APPNAME    = 3,
} USERCLASSTYPE;
```

Elements

USERCLASSTYPE_FULL

The full type name of the class.

USERCLASSTYPE_SHORT

A short name (maximum of 15 characters) that is used for popup menus and the Links dialog box.

USERCLASSTYPE_APPNAME

The name of the application servicing the class and is used in the Result text in dialog boxes.

See Also

IOleObject::GetUserType, **OleRegGetUserType**

VIEWSTATUS Quick Info

The **VIEWSTATUS** enumeration is used in **IViewObjectEx::GetViewStatus** to specify the opacity of the object and the drawing aspects supported by the object.

```
typedef enum tagVIEWSTATUS
{
    VIEWSTATUS_OPAQUE               = 1,
    VIEWSTATUS_SOLIDBKGND           = 2,
    VIEWSTATUS_DVASPECTOPAQUE       = 4,
    VIEWSTATUS_DVASPECTTRANSPARENT  = 8
} VIEWSTATUS;
```

Elements

VIEWSTATUS_OPAQUE

The object is completely opaque. So, for any aspect, it promises to draw the entire rectangle passed to the **IViewObject::Draw** method. If this value is not set, the object contains transparent parts. If it also support DVASPECT_TRANSPARENT, then this aspect may be used to draw the transparent parts only.

This bit applies only to CONTENT related aspects and not to DVASPECT_ICON or DVASPECT_DOCPRINT.

VIEWSTATUS_SOLIDBKGND

The object has a solid background (consisting in a solid color, not a brush pattern). This bit is meaningful only if VIEWSTATUS_OPAQUE is set.

This bit applies only to CONTENT related aspects and not to DVASPECT_ICON or DVASPECT_DOCPRINT.

VIEWSTATUS_DVASPECTOPAQUE

Object supports DVASPECT_OPAQUE. All **IViewObjectEx** methods taking a drawing aspect as a parameter can be called with this aspect.

VIEWSTATUS_DVASPECTTRANSPARENT

The object supports DVASPECT_TRANSPARENT. All **IViewObjectEx** methods taking a drawing aspect as a parameter can be called with this aspect.

See Also

IViewObjectEx::GetViewStatus

OLE Registry Entries

To become functional, OLE servers and containers must add information to the system registry. Most of that information is stored in keys and named values under the **HKEY_LOCAL_MACHINE\SOFTWARE\Classes** and **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE** keys. This section discusses these registry keys and information orequired to support OLE functionality. For general information on the registry, the structure of registry entries, and registry functions, search for Registry topics in the *Microsoft Win32 Programmer's Reference*.

Note that **HKEY_CLASSES_ROOT** is equivalent to **HKEY_LOCAL MACHINE\SOFTWARE\Classes** and that these keys can be used interchangeably. **HKEY_CLASSES_ROOT** also provides compatibility with Windows 3.1 and Windows 95.

Installation and Setup

Your installation/setup program must add information to the registry, and its associated subkeys, if it is to perform any of the three following types of installations:

- Installing an OLE server application.
- Installing an OLE container/server application.
- Installing a container application *that allows linking to its embedded objects*.

In all three cases, you must register OLE 2 library (DLL) information as well as application-specific information.

For information on registering COM servers, see Registering COM Servers..

OLE Registry Functions

For information about OLE registry functions, refer to the following API functions:

- **CoGetTreatAsClass**
- **CoTreatAsClass**
- **OleDoAutoConvert**
- **OleGetAutoConvert**
- **OleSetAutoConvert**
- **SetConvertStg**
- **GetConvertStg**
- **OleRegGetUserType**
- **OleRegEnumFormatEtc**
- **OleRegGetMiscStatus**
- **OleRegEnumVerbs**

Registering OLE 2 Libraries

The OLE 2 libraries require that many internal interfaces be registered. If your installation program installs the OLE 2 libraries on a machine that does not already have them, it should register OLE 2-specific information during installation.

If your installation program does not install OLE 2 libraries when it finds more recent libraries on the user's drive, it should not register the OLE 2 interface information.

Note When checking the version stamp (using VER.DLL) on existing OLE 2 libraries to determine whether or not to replace them on the user's hard drive, check on a per file basis.

Checking Registration During Run Time

An application should check its registration at application load time, noting the following issues:

- If the CLSID(s) that the application services is not present in the registry, the application should register as it does during the original setup.
- If the application's CLSID is present, but has no OLE 2-related information in it, the application should register as it does during the original setup.
- If the path containing server entries (**LocalServer** and **LocalServer32**, **InprocServer** and **InprocServer32**, and **DefaultIcon**) does not point to the location in which the application is currently installed, the application should rewrite the path entries to point to its current location.

Specifying Unknown User Types

It is possible to facilitate product localization by adding a key to the registry. This key allows functions to return a specified string instead of a default value or "Unknown."

The OLE 2 default handler's implementation of **IOleObject::GetUserType** first examines the registry by calling **OleRegGetUserType**. If the object's class is not found in the registry, the User Type from the object's **IStorage** instance is returned. If the class is not found in the object's **IStorage** instance, the string "Unknown" is returned.

By inserting the

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE2
\UnknownUserType = <*usertype*>

key in the registry, the **IOleObject::GetUserType** method returns the value of the string specified by <*usertype*>. This string can be localized for a different language user-type name, instead of using the "Unknown" string, for the User Type.

Conventions Used in Examples

In the registry examples in this appendix, **boldface** indicates a literal standard key or subkey, <*italics*> indicates an application-supplied string or value, and <***boldface-italics***> indicates an application-supplied key or subkey. In the first example, "OLE1ClassName," "OLE1UserTypeName," and "CLSID" are all supplied by the application.

HKEY_LOCAL_MACHINE\SOFTWARE\Classes

The subkeys and named values associated with key contain information about an application that is needed to support OLE functionality. This information includes such topics as supported data formats, compatibility information, programmatic identifiers, distributed com, and controls.

AppID Key

Application identifiers (AppIDs), group the configuration options, a set of named-values, for one or more distributed COM objects into one centralized location in the registry. Distributed COM objects hosted by the same executable are grouped into one **AppID** to simplify the management of common security and configuration settings. The **HKEY_LOCAL_MACHINE\SOFTWARE\Classes** key corresponds to the **HKEY_CLASSES_ROOT**, which definition was retained for compatibility with earlier versions of OLE.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID
{*AppID_value***}***named_value* **=** *value*

Comments

AppIDs are located in a newly created registry key hierarchy. The *AppID_value* is a 128-bit Globally Unique Identifier (GUID) that uniquely identifies the **AppID**.

AppIDs are mapped to executables and classes using two different mechanisms.

Classes indicate their corresponding **AppID** in their **CLSID** key. A named-value "AppID" of type REG_SZ contains the string value corresponding to the *AppID_value* under the **AppID** sub-key. This mapping is used during activation.

Executables are registered under the **AppID** key in a named-value indicating the module name (such as "MYOLDAPP.EXE"). This named-value is of type REG_SZ and contains the stringized AppID associated with the executable. This mapping is used to obtain the default access permissions.

Named Values:

\RemoteServerName = *value*	Sets name of remote server
\ActivateAtStorage = *value*	Configures client to activate on same system as persistent storage
\LocalService = *value*	Sets the application as a Win32 service
\ServiceParameters = *value*	Sets parameters to be passed to a LocalService on invocation
\RunAs = *value*	Sets an application to run only as a given user.
\LaunchPermission = *value*	Sets an ACL that determines who can launch the application
\AccessPermission = *value*	Sets an ACL that determines access

See Also

CLSID key, **OLE Registry Entries**, **Registering COM Servers**

RemoteServerName

A server may install the **RemoteServerName** named-value on client machines to configure the client to request the object be run at a particular machine whenever an activation function is called for which a **COSERVERINFO** structure is not specified.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID
{*AppID_value***}\RemoteServerName =** *server_name*

Remarks

As described in the documentation for the **CLSCTX** enumeration and the **COSERVERINFO** structure, one of the parameters of the distributed COM activation is a pointer to a **COSERVERINFO** structure. When this value is not NULL, the information in the **COSERVERINFO** structure overrides the setting of the **RemoteServerName** key for the function call.

RemoteServerName allows simple configuration management of client applications - they may be written without hard-coded server names, and can be configured by modifying the **RemoteServerName** registry values of the classes of objects they use.

See Also

CLSCTX enumeration, **COSERVERINFO** structure, **CoCreateInstanceEx**, **CoGetInstanceFromFile**, **CoGetInstanceFromIStorage**, **Registering COM Servers**

ActivateAtStorage

Servers can install this named-value on client machines. When this value is not overridden by certain parameters passed to distributed COM activation API functions, the **ActivateAtStorage** named-value configures the client to instantiate objects on the same machine as the persistent state they are using or from which they are initialized.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID
{*AppID_value***}\ActivateAtStorage** = *value*

Remarks

The **ActivateAtStorage** named-value can be set by the server to configure the client to instantiate its objects on the same machine as the persistent state used or from which the objects are initialized. The value is a REG_SZ; any value beginning with **Y** or **y** means that **ActivageAtStorage** should be used.

When a value is set for **ActivateAtStorage**, this becomes the default behavior in calls to the **CoGetInstanceFromFile** and **CoGetInstanceFromIStorage** functions, as well as to the file moniker implementation of **IMoniker::BindToObject**.

In all of these calls, specifying a **COSERVERINFO** structure overrides the setting of **ActivateAtStorage** for the duration of the function call. The caller can pass COSERVERINFO information to **IMoniker::BindToObject** through the **BIND_OPTS2** structure.

The value set for **ActivateAtStorage** is also the default behavior when CLSCTX_REMOTE_SERVER is specified if no registry information for the class is installed on the client's machine. Client applications written to take advantage of **ActivateAtStorage** may therefore require less administration.

The **ActivateAtStorage** capability provides a transparent way to allow clients to locate running objects on the same machine as the data that they use. This reduces network traffic, because the object performs local file-system calls instead of calls across the network.

See Also

CoGetInstanceFromFile, **CoGetInstanceFromIStorage**, **IMoniker::BindToObject**, **COSERVERINFO** structure, **CLSCTX**, **Registering COM Servers**

LocalService

Allows an object to be installed as a Win32 service.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID
{*AppID_value***}\LocalService** = *service_name*

Remarks

In addition to running as a *local server* executable (EXE), a COM object may also choose to package itself to run as a Win32 service when activated by a local or remote client. Win32 services support numerous useful and UI-integrated administrative features, including local and remote starting, stopping, pausing, and restarting, as well as the ability to establish the server to run under a specific user account and Window Station, and optionally interactive with the desktop.

An object written as a Win32 service is installed for use by OLE by establishing a **LocalService** named-value under its **CLSID** key and performing a standard service installation (refer to the Win32 documentation and the SECSVR distributed COM sample application for more information on writing, installing, and debugging Win32 services.). The **LocalService** named-value must be set to the service name - as configured in HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services - as the default REG_SZ value.

Additionally, when **LocalService** is set, any string assigned to the named REG_SZ value, **ServiceParameters**, will be passed on the command line to the service as it is being launched.

For example, the following keys register the c:\mydb\mydb.exe executable to launch as the MyDatabase service when activated remotely. When launched as a service, the process receives "-Service" on its command-line.

```
HKEY_CLASSES_ROOT\APPID\{c5b7ac20-523f-11cf-8117-00aa00389b71}\LocalService
= MyDatabase

HKEY_CLASSES_ROOT\APPID\{c5b7ac20-523f-11cf-8117-00aa00389b71}\
ServiceParameters = -Service
```

The service configuration is preferred in many situations where the capabilities of the local and remote service management APIs and user interface might be useful for the services that the object provides. For example, if the object is long-lived, or readily supports concepts such as starting, stopping, resetting, or pausing, leveraging the existing administrative framework of the service architecture should be an obvious choice.

Services can be dynamically configured using the 'Services' application in the Control Panel, and can be configured to run automatically when the machine boots, or to be launched when requested by a client application.

There are several additional points you should be aware of if you are implementing classes as services:

- The **LocalService** named-value is used in preference to the **LocalServer32** key for local and remote activation requests - if **LocalService** exists and refers to a valid service, the **LocalServer32** key is ignored.
- Currently only a single instance of a Win32 service may be running at a given time on a machine. OLE services must therefore register their class objects on launch using REGCLS_MULTIPLEUSE to support multiple clients.
- OLE services configured to run automatically when a machine boots must include "RPCSS" in their list of dependent services in order to launch and initialize properly.

- Although services may be configured to interact with the interactive user, they are always run in a separate *window-station* and *desktop* from the interactive user. This prevents service processes from being automatically terminated when the interactive user logs off.

See Also

ServiceParameters, **Registering COM Servers**

ServiceParameters

When an object is written as a Win32 service and installed for use by OLE through setting the **LocalService** named value under the **AppID** key, the **ServiceParameters** named-value can be set to pass the given parameter on the command line when the service is launched.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID
{*AppID_value***}\ServiceParameters** = *parameter*

Remarks

The *parameter* value for the **ServiceParameters** named-value is any REG_SZ string value. This string is passed on the command line to the service as it is being launched.

For example, the following keys register the c:\mydb\mydb.exe executable to launch as the MyDatabase service when activated remotely. When launched as a service, the process receives "-Service" on its command-line.

```
HKEY_CLASSES_ROOT\APPID\{c5b7ac20-523f-11cf-8117-00aa00389b71}\LocalService
= MyDatabase

HKEY_CLASSES_ROOT\APPID\{c5b7ac20-523f-11cf-8117-00aa00389b71}\
ServiceParameters = -Service
```

Refer to the Win32 SDK documentation about services and the SECSVR sample Distributed COM application for more information about writing, installing, and debugging Win32 services.

See Also

LocalService, **Registering COM Servers**

RunAs

Configures a class to run under a specific user account when activated by a remote client without being written as a Win32 service. To do this, the RunAs named-value is set for the class to a user-name and optionally a password. These are then used when the Service Control Manager launches its local server process.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID
{*AppID_value***}\RunAs =** *value*

Remarks

The *value* specifies the user name, and must either of the form *username*, *domain\username*, or the string **Interactive User**.

Classes configured to **RunAs** a particular user may not be registered under any other identity, so calls to **CoRegisterClassObject** with this CLSID will fail unless the process was launched by OLE on behalf of an actual activation request.

The user-name is taken from the **RunAs** named-value under the class's **AppID** key. If the user-name is "Interactive User", the server is run in the identity of the user currently logged on and is connected to the interactive desktop.

Otherwise, the password is retrieved from a secret and safe portion of the registry available only to administrators of the machine and to the system. The user-name and password are then used to create a logon-session in which the server is run. When launched in this way, the user runs with its own *desktop* and *window-station*, and does not share window-handles, the clipboard, or other UI elements with the interactive user or other user running in other user accounts.

To establish a password for a **RunAs** class, you must use the DCOMCNFG administrative tool installed in the system directory.

For **RunAs** identities used by DCOM servers, the user account specified in the value must have the rights to log on as a batch job. This right can be added using the NT User Manager, under Policies-User Rights. Click on the *Show Advanced User Rights* box, select *log on as a batch job*, and add the **RunAs** user account.

The **RunAs** value is not used for servers configured to be run as services. COM services that wish to run under an identity other than **LocalSystem** should set the appropriate user name and password using the services control panel applet.

See Also

Registering COM Servers

LaunchPermission

Contains data describing the Access Control List (ACL) of the principals that can start new servers for this class. The **LaunchPermission** named-value may be added under any **AppID** key to limit activation by remote clients of specific classes.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID
{*AppID_value***}\LaunchPermission** = *ACL*

Remarks

The **LaunchPermission** named-value is REG_BINARY. Upon receiving a local or remote request to launch the server of this class, the ACL described by this value is checked while impersonating the client, and its success either allows or disallows the launching of the server. If this value does not exist, the machine-wide **DefaultLaunchPermission** value is checked in the same way as a default to determine if the class code can be launched.

See Also

DefaultLaunchPermission, **Security in COM**

AccessPermission

Contains data describing the Access Control List (ACL) of the principals that can access instances of this class. This ACL is only used by applications that do not call **CoInitializeSecurity**.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID
{*AppID_value***}\AccessPermission** = *ACL*

Remarks

This named-value is a REG_BINARY. It contains data describing the Access Control List (ACL) of the principals that can access instances of this class. Upon receiving a request to connect to an existing object of this class, the ACL is checked by the application being called *while impersonating the caller*. If the access-check fails the connection is disallowed. If this named-value does not exist, the machine-wide **DefaultAccessPermission** ACL is tested in an identical manner (see above) as a default to determine if the connection is to be allowed.

See Also

DefaultAccessPermission, **Security in COM**

CLSID Key

A CLSID is a globally unique identifier that identifies an OLE class object. If your server or container allows linking to its embedded objects, then you need to register a CLSID for each supported class of objects.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID = <*CLSID*>

Value Entries

<CLSID>

Specifies a name that can be displayed in the user interface.

Remarks

The CLSID key contains information used by the default OLE handler to return information about a class when it is in the running state.

To obtain a CLSID for your application, you can use the UUIDGEN.EXE found in the \TOOLs directory of the OLE 2 Toolkit, or use **CoCreateGuid**.

The CLSID is a 128 bit number, spelled in hex, within a pair of brarces.

SubKeys and Named Values

***CLSID* = <*CLSID*>**

\<*CLSID*>	Human readable name
\AppID	Application identifiers
\AutoConvertTo	Automatic object class conversion
\AutoTreatAs	Assigns the TreatAs value.
\AuxUserType	Identifies object as a control
\Control	Displayable application name.
\Conversion	Conversion used by the Convert dialog
\DataFormats	Formats supported by applications.
\DefaultIcon	Provides default icon information
\InprocHandler	Registers a 16-bit handler DLL.
\InprocHandler32	Registers a 32-bit handler DLL.
\InprocServer	Registers a 16-bit in-process server DLL
\InprocServer32	Registers a 32-bit in-process server DLL
\Insertable	Indicates object is insertable in OLE 2 applications
\Interface	Associates interface name with IID
\LocalServer	Full path to a 16- or 32-bit application
\LocalServer32	Full path to a 32-bit application
\MiscStatus	Default status used for all aspects
\ProgID	Programmatic identifier for a class
\ToolBoxBitmap32	Module name and resourdeID for a 16 x 16 bitmap
\TreatAs	OILE1 / OLE 2 compatibility.
\Verb	Verbs associated with an application
\Version	Version number of the control

See Also

CoCreateGuid

<CLSID>

A globally unique identifier (GUID) used to map information about a component class.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\<*CLSID***>=** *HumanReadableName*

Value Entries

HumanReadableName

Specifies a name that can be displayed in the user interface.

Remarks

COM uses the information mapped by the CLSID to locate and create an instance the object associated with the CLISID.

See Also

CoCreateGuid

AppID

Associate an AppID with a CLSID.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\AppID = *<CLSID>*

Value Entries

<CLSID>

The AppID to associate with this CLSID.

See Also

AppID

AutoConvertTo

Specifies the automatic conversion of a given class of objects (*ClsidOld*) to a new class of objects (*pClsidNew*).

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\AutoConvertTo = <*CLSID*>

Value Entries

<*CLSID*>

The class identifier of the object to which a given object or class of objects should be converted.

Remarks

This key is typically used to automatically convert files created by an older version of an application to a newer version of the application.

See Also

OleGetAutoConvert, **OleDoAutoConvert**, **OleSetAutoConvert**

AutoTreatAs

Automatically sets the CLSID for the **TreatAs** key to the specified value.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\AutoTreatAs = <*CSID***>**

Value Entries

MCLSID>

The *CLSID* that will automatically be assigned to the **TreatAs** entry.

See Also

CoTreatAsClass

AuxUserType

Specifies an application's short display name and application names.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\AuxUserType
\2 = <*ShortDisplayName*>
\3 = <*ApplicationName*>

Value Entries

<*ShortDisplayName*>

Specifies an application's short display name, used in menus, including pop-ups.

<*ApplicationName*>

Specifies an application name, used in the Results field of the Paste Special dialog box.

Remarks

The recommended maximum length for the short display name is 15 characters. For example:

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \AuxUserType\2 = In-Place Outline
```

The application name should contain the actual name of the application (such as "Acme Draw 2.0"). For example:

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \AuxUserType\3 = Ole 2 In-Place Server
```

See Also

IOleObject::GetUserType

Control

Identifies an object as an OLE Control.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\Control

Remarks

This optional entry is used by containers to fill in dialog boxes. The container uses this subkey to determine whether to include an object in a dialog box that displays OLE Controls.. A control can omit this entry if it is only designed to work with a specific container and thus does not wish to be inserted in other containers.

Conversion

Used by the Convert Dialog to determines the formats an application can read and write.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\Conversion
\Readable
\Main = <*forma***>***t, ...*
\ReadWritable
\Main = <*format***>***, ...*

Value Entries

<format>

The file format an application can read (convert from).

<format>

The file format an application can read and write (activate as).

Remarks

Conversion information is used by the Convert dialog box to determine what formats an application can read and write. A comma-delimited file format is indicated by a number if it is one of the Clipboard formats defined in WINDOWS.H. A string indicates the format is not one defined in WINDOWS.H (private). In this case, the readable and writable format is CF_OUTLINE (private).

The following is a **\Readable** entry:

```
KEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \Conversion\Readable\Main = Outline,1
```

The following is a **\ReadWritable** entry:

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \Conversion\Readwritable\Main = Outline,1
```

DataFormats

Specifies the default and main data formats supported by an application.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\DataFormats
\DefaultFile = <*defaultfile/objectformat***>**
\GetSet
\ *n* = <*default formats for EnumFormatEt***>**

Value Entries

<default file/object format>

Specifies the default, main file or object format for objects of this class.

<default formats for EnumFormatEtc>

Specifies a list of formats for default implementations of **EnumFormatEtc**, where **<*n*>** is a zero-based integer index. For example, **<*n*>** = <*format* ,*aspect*, *medium*, *flag*>, where *format* is a clipboard format, *aspect* is one or more members of **DVASPECT**, *medium* is one or more members of **TYMED**, and *flag* is one or more members of **DATADIR**.

Remarks

Information associated with this entry is used by: **IDataObject::GetData**, **IDataObject::SetData** and **IDataObject::EnumFormatEtc** methods.

The values defined in the following example entry are CF_TEXT, DVASPECT_CONTENT, TYMED_HGLOBAL, and DATADIR_GET | DATADIR_SET.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \DataFormats\GetSet\0 = 1,1,1,3
```

The values defined in the following entry are: CF_METAFILEPICT DVASPECT_CONTENT, TYMED_MFPICT, DATADIR_GET.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \DataFormats\GetSet\1 = 3,1,32,1
```

The values defined in the following entry are: 2 = cfEmbedSource, DVASPECT_CONTENT, TYMED_ISTORAGE, and DATADIR_GET.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \DataFormats\GetSet\2 = Embed Source,1,8,1
```

The values defined in the following entry are: 3 = cfOutline, DVASPECT_CONTENT, TYMED_HGLOBAL, and DATADIR_GET | DATADIR_SET.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \DataFormats\GetSet\3 = Outline,1,1,3
```

The following entry declares that the default File Format supported by this application is CF_OUTLINE.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \DataFormats\DefaultFile = Outline
```

See Also

IDataObject::GetData, IDataObject::SetData, IDataObject::EnumFormatEtc

DefaultIcon

Provides default icon information for iconic presentations of objects.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\DefaultIcon = <*full path to exe, resoruce id***>**

Value Entries

<full path to exe, resource id>

Specifies the full path to the executable name of the object application and the resourceID of the icon within the executable.

Remarks

DefaultIcon identifies the icon to use when, for example, a control is minimized to an icon. This entry contains the full path to the executable name of the server application and the resourceID of the icon within the executable. Applications can use the information provided by **DefaultIcon** to obtain an icon handle with **ExtractIcon**.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \DefaultIcon = c:\samp\isvrotl.exe,0
```

See Also

ExtractIcon

InprocHandler

Specifies whether or not an application uses a custom handler.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\InprocHandler = <*handler.dll***>**

Value Entries

<handler.dll>

Specifies the custom handler used by the application.

Remarks

If A custom handler is not used, the entry should be set to OLE2.DLL, as shown in the following example.

```
HKEY_-CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \InprocHandler = ole2.dll
```

If a container is searching the registry for an InprocHandler, the 16-bit version has priority with a 16-bit container, and the 32-bit version has priority with a 32-bit container.

See Also

InprocHandler32

InprocHandler32

Specifies whether or not an application uses a custom handler.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\InprocHandler32 = <*handler.dll***>**

Value Entries

<handler.dll>

Specifies the custom handler used by the application.

Remarks

If A custom handler is not used, the entry should be set to OLE32.DLL, as shown in the following example.

```
HKEY_-CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \InprocHandler = ole32.dll
```

If a container is searching the registry for an InprocHandler, the 16-bit version has priority with a 16-bit container, and the 32-bit version has priority with a 32-bit container.

See Also

InprocHandler

InprocServer

Specifies the path to the inprocess-server DLL.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\InprocServer = <*full path*>

Value Entries

<full path>

Specifies the path to the inprocess server DLL.

Remarks

The InprocServer entry is relatively rare for insertable classes.

If a container is searching the registry for an InprocServer, the 16-bit version has priority with a 16-bit container, and 32-bit version has priority with a 32-bit container.

See Also

InprocServer32

InprocServer32

Registers a 32-bit in-process server DLL and specifies the threading model.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\InprocServer32 = *<path to 32-bit inproc server dll>*
\ThreadingModel = *<threading model>*

Value Entries

<path to 32-bit inproc server dll>

Specifies the path to the 32-bit inproc server.

<threading model>

Specifies the threading model. In-process servers do not **CoInitialize** call or **CoInitializeEx**; they must use the registry to specify an applications threading model. Threading models used for this purpose are:

- ThreadingModel = no value specified: Supports single threading model.
- ThreadingModel=Apartment. Supports apartment model.
- ThreadingModel=Both. Supports apartment model and free threading.
- ThreadingModel=Free. Supports only free threading...

Currently the ThreadingModel value must be the same for all objects provided by an inproc server.

Remarks

For a 32-bit InprocServer, the required entries are InprocHandler32, InprocServer, InprocServer32, and Insertable. Note that InprocServer entry provides backward compatibility. If it is missing, the class will still work, but can't be inserted in 16-bit applications.

If a container is searching the registry for an InprocServer, the 16-bit version has priority with a 16-bit container, and 32-bit version has priority with a 32-bit container.

The required entries for 32-bit InprocServers are InprocHandler32, InprocServer, InProcServer32, and Insertable. Note that LocalServer and InprocServer entries provide backward compatibility. If they are missing, the class will work, but cannot be inserted into 16-bit applications.

See Also

InprocServer, **CoInitialize**, **CoInitializeEx**

Insertable

Indicates that objects of this class should appear in the Insert Object dialog box's list box when used by OLE 2 container applications.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\Insertable

Remarks

This key is a required entry for 32-bit OLE applications whose objects can be inserted into existing 16-bit applications. Existing 16-bit applications look in the registry for this key, which informs the application that the server supports embeddings. If the Insertable key exists, 16-bit applications may also attempt to verify that the server exists on the machine. 16-bit applications typically will retrieve the value of the LocalServer key from the class and check to see if it is a valid file on the system. Therefore, for a 32-bit application to be insertable by a 16-bit application, the 32-bit application should register the LocalServer subkey in addition to registering LocalServer32.

Used with controls, this entry indicates that an object can act only as an in-place embedded object with no special control features. Objects that have this key appear in the Insert Object dialog box fo their container. When used with controls, this entry also indicates the control has been tested with non-control containers. This entry is also optional and can be omitted when a control is not designed to work with older containers that do not understand controls.

Note that this key is not present for internal classes like the moniker classes.

See Also

ProgID

Interface

An optional entry that specifies all interface IDs (IIDs) supported by the associated class.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\Interface

\<*IID*> = <*name of interface1***>**
\<*IID*> = <*name of interface2***>**
\...

Value Entries

<name of interface1>, <name of interface2>, ...

Interfaces supported by this class.

Remarks

If an interface name is not present in this list, then the interface can never be supported by an instance of this class.

See Also

Interface

LocalServer

Specifies the full path to a 16-bit local server application.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\LocalServer = <*full path***>**

Value Entries

<full path>

Specifies the full path to the local server, and can include command-line arguments.

Remarks

OLE 2 appends the "-Embedding" flag to the string, so the application that uses flags must parse the whole string and check for the -Embedding flag.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \LocalServer = c:\samp\isvrotl.exe
```

To run an OLE object server in a separate memory space (Windows NT 3.5 and above only), change the LocalServer key in the registry for the CLSID to the following:

```
cmd /c start /separate <path.exe
```

If a container is searching the registry for a local server, a 32-bit local server has priority over a 16-bit local server.

See Also

LocalServer32

LocalServer32

Specifies the full path to a 32-bit local server application.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\LocalServer32 = *<full path>*

Value Entries

<full path>

Specifies the full path to the 32-bit local server, and can include command-line arguments.

Remarks

OLE 2 appends the "-Embedding" flag to the string, so the application that uses flags will need to parse the whole string and check for the -Embedding flag.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \LocalServer32 = c:\samp\isvrotl.exe
```

When OLE starts a 32-bit local server, the server must register a class object within an elapsed time set by the user. By default, the elapsed time value must be at least five minutes, in milliseconds, but cannot exceed the number of milliseconds in 30 days. Applications typically should not set this value which is in the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE2\ServerStartElapsedTime entry.**

The required entries for 32-bit local servers are InprocHandler32, LocalServer, LocalServer32, and Insertable.

If a container is searching the registry for a local server, a 32-bit local server has priority over a 16-bit local server.

See Also

LocalServer

MiscStatus

Specifies how to create and display an object.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\MiscStatus = <*object description***>**
*****n* **=** *value*

Value Entries

object description

The value or combination of values from **OLEMISC** that describe the object.

value

Specifies a value from the OLEMISC enumeration.

Remarks

Information used to describe objects is found in the **OLEMISC** enumeration.

The following is an example of a **MiscStatus** entry. Refer to the **IOleObject::GetMiscStatus** method description for information on the different settings.

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \MiscStatus = 2
```

See Also

IOleObject::GetMiscStatus, **OLEMISC**

ProgID

Associates a ProgID with a CLSID.

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\ProgID = <*programatic identifier***>**

Value Entries

<programatic identifier>

Specifies the ProgID: <*vendor*>.<*component*>.<version>.

Remarks

Every insertable object class has a <*ProgID*>. For information on creating a <*ProgID*>, see the <*ProgID*> key.

See Also

VersionIndependentProgID, <*ProgID*>

ToolBoxBitmap32

Identifies the module name and resourdeID for a 16 x 16 bitmap to use for the face of a toolbar or toolbox button.

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\ToolBoxBitmap32 = <*filename*>.<*ext*>, *resourceID*

Value Entries

<*filename*>.<*ext*>, *resourceID*>

Specifies the module name and the resorucID for the bitmap.

Remarks

The standard Windows icon size is too large to be used for this purpose. This specifically supports control containers that have a design mode in which one selects controls and places them on a form being designed. For example, in Visual Basic, the control's icon is displayed in the Visual Basic Toolbox during design mode.

TreatAs

Specifies the *CLSID* of a class that can emulate the current class.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\TreatAs = <*CLSID***>**

Value Entries

<CLSID>

*Cla*ss that is to perform the emulation.

Remarks

Emulation is the ability of one application to open and edit an object of a different class, while retaining the original format of the object.

See Also

CoTreatAsClass, **CoGetTreatAsClass**, **AutoTreatAs**, **IOleObject::EnumVerbs**

Verb

Specifies the verbs to be registered for an application.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\Verb
\1 = <*verb1*>
\2 = <*verb2*>
\3 =

Value Entries

<verb1>, <verb2>, ...

Each *value* specifies a verb, and its associated menu and verb flags. Each verb is specified by its own **NamedValue** / *Value* pair, as in **verb number** = <*name*, *menu flag*, *verb flag*> For example:

\Verb
0 = &Edit, 0, 2 // primary verb; on menu, possibly dirties object,
1 = &Play, 0, 3 // other verb; on menu; leaves object clean
-3 = Hide, 0, 1 // pseudo verb, hides window; not on menu, opt.
-2 = Open, 0, 1 // pseudo verb, opens in sep. window; not on menu, opt.
-1 = Show, 0, 1 // pseudo verb, show in preferred state; not on menu, opt.

See **IOleObject::DoVerb** for general information about verbs, descriptions of OLE predefined verbs, and positive and negative verbs, plus other material.

Remarks

Verbs must be numbered consecutively. The first value after the verb string describes how the verb is appended by an **AppendMenu** function call.

The second value indicates whether the verb will dirty the object. It also indicates whether the verb should appear in the menu, as defined by OLEVERBATTRIB enumeration, used by the OLEVERB structure.

For still more information, see **IOleObject::EnumVerbs** and **OleRegEnumVerbs**.

Following are two example entries:

Verb 0: "Edit", MF_UNCHECKED | MF_ENABLED, no OLEVERATTRIB flags:

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \Verb\0 = &Edit,0,0
```

Verb 1: "Open", MF_UNCHECKED | MF_ENABLED, no OLEVERATTRIB flags:

```
HKEY_CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \Verb\1 = &Open,0,0
```

See Also

IOleObject::EnumVerbs, **OleRegEnumVerbs**, **AppendMenu**, **OLEVERB**, **OLEVERBATTRIB**

Version

Specifies the version number of the control.

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID
\Version = <*VersionNumber***>**

Value Entries

<ersionNumber>

The version number of the control.

Remarks

The version number should match the version of the type library associated with the control.

ProgID Key

A ProgID, or programmatic identifier, is a registry entry that can be associated with a CLSID. The format of a ProgID is <*Vendor*>.<*Component*>.<*Version*>, separated by periods and with no spaces, as in Word.Document.6. Like the CLSID, the ProgID identifies a class, but with less precision.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\<*ProgID***> =**

You can use a ProgID in programming situations where it is not possible to use a CLSID. ProgIDs should not appear in the user interface. ProgIDs are not guaranteed to be unique so they can be used only where name collisions are manageable.

The <*ProgID*> must:

- Have no more than 39 characters.
- Contain no punctuation (including underscores) except one or more periods.
- Not start with a digit.
- Be different from the class name of any OLE 1 application, *including the OLE 1 version of the same application*, if there is one.

Since the <*ProgID*> should not appear in the user interface, you can obtain a displayable name by calling **IOleObject::GetUserType**. Also, see **OleRegGetUserType**>

The value of the <*ProgID*> is a human readable name such as Microsoft Word Document, and is displayed in dialog boxes.

SubKeys and Named Values

\<*ProgID***> =** <*HumanReadableNamen*>

CLSID	Object's CLSID
Insertable	Indicates that class is insertable in OLE 2 containers
Protocol	Indicates class is insertable in OLE 1 container
Shell	Windows 3.1 File Manager information

See Also

IOleObject::GetUserType, **OleRegGetUserType**

CLSID

Associates a ProgID with a CLSID.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\ProgID
\CLSID = <*CLSID*>

Value Entries

<CLSID>

The object's CLSID.

See Also

GetClassFile

Insertable

Indicates that this class is insertable in OLE 2 containers.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\ProgID
\Insertable

Remarks

This is a required entry for objects that are insertable in OLE 2 containers.

Protocol

Indicates that this OLE 2 class is insertable in OLE 1 containers.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\ProgID
\Protocol
\StdFileEditing
\Server = <*full path to OLE 2 server app***>**
\Verb =
\0 = <*verb value0***>**
\1 = <*verb value***>**
*****, ...

Value Entries

<full path to OLE 2 server app>

Specifies the full path to the OLE 2 server application.

<verb value>

The primary verb; must start with zero.

<verb value>, ...

Additional verb, numbered consecutively.

Remarks

The StdFileEditing entry is specifies OLE 1 compatibility information. It should be present only if, objects of this class are insertable in OLE 1 containers.

Shell

Provides Windows 3.1 shell printing and File Open information.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\ProgID
\Shell
\Open
\command = <*path to application exe* %1>
\Print
\command = <*path to application exe* %1>

Value Entries

<*path to application exe* %1>

Specifies the path to the executable.

Remarks

These entries should provide the path and filename of the application. The examples provided here are simple entries. More complex entries could contain DDE entries.

```
HKEY_CLASSES_ROOT\OLE2ISvrOtl\Shell\Print\Command =
    c:\svr\isvrotl.exe %1
HKEY_CLASSES_ROOT\OLE2ISvrOtl\Shell\Open\Command =
    c:\svr\isvrotl.exe %1
```

VersionIndependentProgID Key

The values associated with this key associate a ProgID with a CLSID. Used to determine the latest version of an object application.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes
\<*VersionIndependentProgID*> = <*Vendor*>.<*Component*>
\CLSID = <*CLSID*>
\CurVer = <*ProgID*>

Value Entries

<*Vendor*>.<*Component*>

The name of the latest version of the object application.

<*CLSID*>

The CLSID of the newest installed version of the class.

<*ProgID*>

The <*ProgID*> of the newest installed version of the class.

Remarks

The format or the <*VersionIndependentProgID*> is <*Vendor*>.<*Component*>, separated by periods, no spaces, and no version number. The version-independent ProgID, like the ProgID, can be registered with a human readable name.

Applications must register a version-independent programmatic identifier under the VersionIndependentProgID key. The <*VersionIndependentProgID*> refers to the application's class, and does not change from version to version, instead remaining constant across all versions, for example, Microsoft Word Document. It is used with macro languages and refers to the currently installed version of the application's class. The <*VersionIndependentProgID*> must correspond to the name of the latest version of the object application.

The <*VersionIndependentProgID*> is used when, for example, a container application creates a chart or table with a toolbar button. In this situation the application can use the <*VersionIndependentProgID*> to determine the latest version of the needed object application.

The <*VersionIndependentProgID*> is stored and maintained solely by application code. When given the VersionIndependentProgID, the **CLSIDFromProgID** function returns the CLSID of the current version.

You can use **CLSIDFromProgID** and **ProgIDFromCLSID** to convert between these two representations.

You can use **IOleObject::GetUserType** or **OleRegGetUserType** to change the identifier to a displayable string.

If A custom handler is not used, the entry should be set to OLE32.DLL, as shown in the following example.

```
HKEY_-CLASSES_ROOT\CLSID\{00000402-0000-0000-C000-000000000046}
    \InprocHandler = ole32.dll
```

In addition to the preceding registry entry, you should add the following corresponding entry under the CLSID key:

```
\CLSID
    \<CLSID>  = <human readable name>
        \<VersionIndependentProgID> = human readable name
```

See Also

ProgID, CLSIDFromProgID, ProgIDFromCLSID, IOleObject::GetUserType, OleRegGetUserType

File Extension Key

Associates a file extension with a ProgID, indicating that an OLE 2 application can handle requests from the Windows 3.1 File Manager.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes
\<.*ext*> = <*ProgID*>

Value Entries

<ProgID>

The ProgID associated with

Remarks

The information contained in the file extension key is used by both File Manager and File Monikers. **GetClassFile** uses the file extension key to supply the associated CLSID.

See Also

GetClassFile

(Non-Compound) FileType Key

Used by **GetClassFile** to match patterns against various file bytes in a non-compound file.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes
\FileType
\<*CLSID*>
***<n>** = <*offset*, *cb*, *mask*, *value*>

Value Entries

<*offset*, *cb*, *mask*, *value*>

Offset and *cb* are limited to 16 bits. *Offset* is from the beginning or end of the file, with a negative value for offset being interpreted from the end of the file; *cb* is a the length in bytes. Together, these values represent a particular byte range in the file. *mask* is a bit mask used to perform a logical AND operation, using the byte range spedified by *offsert* and *cb*. The result of the AND operation is compared with If *mask* is omitted, it is assumed to all ones. *offset* and *cb* are decimal unless preceded by "0x". *mask* and *pattern* are always hex.

Remarks

Entries under the FileType key are used by the **GetClassFile** function to match patterns against various file bytes in a non-compound file. FileType has *CLSID* subkeys, each of which has a series of subkeys \ 0, \1, \2, ... These values contain a pattern that, if matched, should yield the indicated *CLSID*. See also the **GetClassFile** function.

Following are examples of FileType entries.
\0 = 0, 4, FFFFFFFF, ABCD1234
where the first 4 bytes must be ABCD12 34, in that order, or
\1 = 0, 4, FFFFFFFF, 9876543
where they must match 9876543, or .
\2 = -4, 4, FEFEFEFE
where the last four bytes in the file must be FEFEFEFE.

See Also

GetClassFile, **File Extension**

Interface Key

Registers new interfaces by associating an interface name with an interface ID (IID).

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface

If your application adds a new interface, the Interface key must be completed for OLE 2 to register the new interface. There must be one IID subkey for each new interface.

Note that you must use **ProxyStubCLSID32** because the IID-to-CLSID mapping may be different for 16- and 32-bit interfaces. The IID-to-CLSID depends on the way the interface proxies are packaged into a set of proxy DLLs.

SubKeys and Named Values

\Interface

<*IID***>**	Interface identifier
\BaseInterface	Interface derived from
\NumMethods	Number of methods
\ProsyStubCLSID	Maps IID to CLSiD (16-bit DLLs)
\ProsyStubCLSID32	Maps IID to CLSiD (32-bit DLLs)

See Also

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\Interface

<*IID*>

Associates an interface ID (IID) with a textual name for the interface.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes
\Interface
\<IID> = *name of interface*

Value Entries

<name of interface>

Textual name for a given interface. For example:
\{00000112-0000-0000-C000-0000000000-46} = **IOleObject**

Remarks

If your application adds a new interface, the interface key must be completed for OLE 2 to register the new interface. There must be one entry for each new interface.

BaseInterface

Identifies the interface from which the current interface is derived.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface
\BaseInterface = <*name of interface***>**

Value Entries

<*name of interface*>

Identifies the name of the interface from which the current interface is derived.

NumMethods

Contains the number of interfaces in the associated interface.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes
\Interface
\NumMethods = **<***number of methods***>**

Value Entries

<*number of methods*>

Specifies the number of methods in the interface.

ProxyStubClsid

Maps an IID to a CLSID in 16-bit proxy DLLs.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes
\Interface
\ProxyStubClsid = <*CLSID***>**

Value Entries

<CLSID>

Specifies the CLSID to map the IID to.

Remarks

If you add interfaces, you must use this entry to register them (16-bit systems) so that OLE can find the appropriate remoting code to establish interprocess communication.

See Also

ProxyStubClsid32

ProxyStubClsid32

Maps an IID to a CLSID in 32-bit proxy DLLs.

Registry Entry

HKEY_LOCAL_MACHINE\SOFTWARE\Classes
\Interface
\ProxyStubClsid32 = <*CLSID***>**

Value Entries

<CLSID>

Specifies the CLSID to map the IID to.

Remarks

This is a required entry since the IID-to-CLSID mapping may be different for 16- and 32-bit interfaces. The IID-to-CLSID mapping depends on the way the interface proxies are packaged into a set of proxy DLLs.

If you add interfaces, you must use this entry to register them (32-bit systems)so that OLE can find the appropriate remoting code to establish interprocess communication.

See Also

ProxyStubClsid, **IMarshal**

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE

The named-values under the HKEY_LOCAL_MACHINE\Software\Microsoft\OLE key control Distributed COM's default launch and access permission settings and call-level security capabilities for applications that do not call **CoInitializeSecurity**. Only machine administrators and the system have full-access to this portion of the registry. All other users have only read-access.

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE*named_value* **=** *value*

Named Values:

EnableDCOM = *value*	Sets the global activation policy for the machine
DefaultLaunchPermission = *value*	Defines default ACL for the machine
DefaultAccessPermission = *value*	Defines default access permission list for the machine
LegacyAuthenticationLevel = *value*	Sets the default authentication level
LegacyImpersonationLevel = *value*	Sets the default impersonation level
LegacyMutualAuthentication = *value*	Determines if mutual authentication is enabled
LegacySecureReferences = *value*	Determines if AddRef/Release invocations are secure

See Also

Security in COM

EnableDCOM

The **EnableDCOM** controls the global activation policies of the machine. Only machine administrators and the system have full-access to this portion of the registry. All other users have only read-access.

Registry Entry

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE
EnableDCOM = *value*

Remarks

This named-value is a REG_SZ, and has two possible values.

N (or **n**)

No remote clients may launch servers or connect to objects on this machine. Local launching of class code and connecting to objects is allowed on a per-class basis according to the value and access permissions of the class's AppID\{…}\LaunchPermission key and the global DefaultLaunchPermission key. See the description of these keys, below.

Y (or **y**)

Launching of servers and connecting to objects by remote clients is allowed on a per-class basis according to the value and access permissions of the class's **LaunchPermission** named-value and the global **DefaultLaunchPermission** named-value.

See Also

LaunchPermission, **DefaultLaunchPermission**, **Security in COM**

DefaultLaunchPermission

Defines the Access Control List (ACL) of the principals that can launch classes that do not specify their own ACL through the **LaunchPermission** named-value

Registry Entry

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE*named_value*

Remarks

The value for the **DefaultLaunchPermission** named-value is a REG_BINARY that contains the Access Control List (ACL) of the principals who can launch classes on the current system. If the **LaunchPermission** named-value is set for a server, it takes precedence over the DefaultLaunchPermission named-value. Upon receiving a local or remote request to launch a server whose APPID key has no LaunchPermission value of its own, the ACL described by this value is checked while impersonating the client, and its success either allows or disallows the launching of the class code.

This entry supports a simple level of centralized administration of the default launching access to otherwise unadministered classes on a machine. For example, an administrator might use the DCOMCNFG tool to configure the system to allow read-access only for *power-users* of the machine. OLE would therefore restrict requests to launch class code to members of the *power-users* group. The administrator could subsequently configure launch permissions for individual classes to grant the ability to launch class code to other groups or individual users as needed.

The access-permissions in this named-value default to the following:

- machine-administrators: allow-launch
- SYSTEM: allow-launch
- INTERACTIVE: allow-launch

See Also

LaunchPermission, **Registering COM Servers**

DefaultAccessPermission

Sets the Access Control List (ACL) of the principals that can access classes for which there is no **AccessPermission** setting. This ACL is only used by applications that don't call **CoInitializeSecurity** and do not have an **AccessPermission** value under their **AppID**.

Registry Entry

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\DefaultAccessPermission = *ACL*

Remarks

The **DefaultAccessPermission** is a named-value that is set to a REG_BINARY that contains data describing the Access Control List (ACL) of the principals who can access classes for which there is no **AccessPermission** named-value. In this case, the server checks the ACL described by this value *while impersonating the caller* that is attempting to connect to the object, and its success determines if the access is allowed or disallowed. If the access-check fails, the connection to the object is disallowed. If this named value does not exist, only the ID of the server and local system are allowed to call the server.

This key supports a simple level of centralized administration of the default connection access to running objects on a machine.

The access-permissions on this key default to the following:

- machine-administrators: allow-access
- SYSTEM: allow-access
- INTERACTIVE: allow-access

See Also

AccessPermission, **Registering COM Servers**

LegacyAuthenticationLevel

Sets the default authentication level for applications that do not call **CoInitializeSecurity**.

Registry Entry

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE
LegacyAuthenticationLevel= *default_authentication_level*

Remarks

LegacyAuthenticationLevel is a named-value is a REG_WORD that sets the default level of authentication for all applications which do not call **CoInitializeSecurity**. Values are from 1 through 6, and correspond to the **RPC_C_AUTHN_LEVEL_*xxx*** constants:

Value	Meaning
1	RPC_C_AUTHN_LEVEL_NONE
2	RPC_C_AUTHN_LEVEL_CONNECT
3	RPC_C_AUTHN_LEVEL_CALL
4	RPC_C_AUTHN_LEVEL_PKT
5	RPC_C_AUTHN_LEVEL_PKT_INTEGRITY
6	RPC_C_AUTHN_LEVEL_PKT_PRIVACY

When this named-value is not present, the default authentication level established by the system is 2 (RPC_C_AUTHN_CONNECT).

See Also

CoInitializeSecurity, **RPC_C_AUTHN_LEVEL_*xxx***, **Registering COM Servers**

LegacyImpersonationLevel

Sets the default level of impersonation for applications that do not call **CoInitializeSecurity**.

Registry Entry

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\LegacyImpersonationLevel**=**
default_impersonation_level

Remarks

The **LegacyImpersonationLevel** named-value is a REG_WORD that sets the default level of impersonation. The values, from 1 through 4, correspond to the values of the **RPC_C_IMP_LEVEL_*xxx*** *constants.*

Value	Meaning
1	RPC_C_IMP_LEVEL_ANONYMOUS
2	RPC_C_IMP_LEVEL_IDENTIFY
3	RPC_C_IMP_LEVEL_IMPERSONATE
4	RPC_C_IMP_LEVEL_DELEGATE

When this named-value is not present, the default impersonation level established by the system is 2 (RPC_C_IMP_LEVEL_IDENTIFY). NTLMSSP on Windows NT 4.0 supports only RPC_C_IMP_LEVEL_IDENTIFY and RPC_C_IMP_LEVEL_IMPERSONATE.

See Also

RPC_C_IMP_LEVEL_*xxx*, **Registering COM Servers**

LegacyMutualAuthentication

Determines whether mutual authentication is enabled.

> **Note** Mutual authentication is not supported and is not available by default with Windows NT 4.0. This named-value is only useful if the network administrator installs a security provider that supports mutual authentication.

Registry Entry

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\LegacyMutualAuthentication= *value*

Remarks

The **LegacyMutualAuthentication** named-value is a REG_SZ that provides the default setting for use of mutual authentication for all applications that do not call **CoInitializeSecurity**. Values of "**Y**" or "**y**" indicate that mutual authentication is enabled. Any other value or the lack of this named-value implies that mutual authentication is disabled.

See Also

CoInitializeSecurity, **Registering COM Servers**

LegacySecureReferences

Determines whether AddRef/Release invocations are secure for applications that do not call **CoInitializeSecurity**.

Registry Entry

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\LegacySecureReferences= *ACL*

Remarks

HKEY_LOCAL_MACHINE\Software\Microsoft\OLE

The **LegacySecureReferences** named-value is a REG_SZ that provides the default setting for securing **IUnknown::AddRef** and **IUnknown::Release** method invokations for all applications that do not call **CoInitializeSecurity**. Values of "**Y**" or "**y**" indicate that **AddRef**/**Release** is secured. Any other value or the lack of this named-value implies that AddRef/Release is not secured. Enabling secure references slows COM.

See Also

CoInitializeSecurity, **Registering COM Servers**

Glossary

A

Activation
The process of loading an object in memory, which puts it into its running state. *See also* Binding.

Active state
A COM object that is in running state and has a visible user interface. *See also* Loaded state, Object state, Passive state, and Running state.

Absolute moniker
A moniker that specifies the absolute location of an object. An absolute moniker is analogous to a full path. *See also* Moniker.

Advisory holder
A COM object that caches, manages, and sends notifications of changes to container applications' advisory sinks. *See also* Advisory sink.

Advisory sink
A COM object that can receive notifications of changes in an embedded object or linked object because it implements the **IAdviseSink** or **IAdviseSink2** interface. Containers that need to be notified of changes in objects implement an advisory sink. Notifications originate in the server, which uses an advisory holder object to cache and manage notifications to containers. *See also* Advisory holder, Container application, and Object handler.

Aggregate object
A COM object that is made up of one or more other COM objects. One object in the aggregate is designated the controlling object, which controls which interfaces in the aggregate are exposed and which are private. This controlling object has a special implementation of **IUnknown** called the controlling **IUnknown**. All objects in the aggregate must pass calls to **IUnknown** methods through the controlling **IUnknown**. *See also* Aggregation.

Aggregation
A composition technique for implementing COM objects. It allows you to build a new object by reusing one or more existing objects' interface implementations. The aggregate object chooses which interfaces to expose to clients, and the interfaces are exposed as if they were implemented by the aggregate object. Clients of the aggregate object communicate only with the aggregate object. *See also* Aggregate object. *Contrast with* Containment.

Ambient property
A run-time property that is managed and exposed by the container. Typically, an ambient property represents a characteristic of a form, such as background color, that needs to be communicated to a control so that the control can assume the look and feel of its surrounding environment. *See also* Run-time property.

Anti-moniker

The inverse of a file, item, or pointer moniker. An anti-moniker is added to the end of a file, item, or pointer moniker to nullify it. Anti-monikers are used in the construction of relative monikers. *See also* Relative Moniker.

Artificial reference counting

A technique used to safeguard an object before calling a function or method that could prematurely destroy it. A program calls **IUnknown::AddRef** to increment the object's reference count before making the call that could free the object. After the function returns, the program calls **IUnknown::Release** to decrement the count.

Asynchronous binding

A type of binding in which it is necessary for the process to occur asynchronously to avoid performance degradation for the end user. Typically, asynchronous binding is used in distributed environments such as the World Wide Web. OLE supports asynchronous moniker classes and callback mechanisms that allow the process of locating and initializing an object in a distributed environment to occur while other operations are being carried out. *See also* Asynchronous moniker, and Binding.

Asynchronous call

A call to a function that is executed separately so that the caller can continue processing instructions without waiting for the function to return. *Contrast with* Synchronous call.

Asynchronous moniker

A moniker that supports asynchronous binding. For example, instances of the system-supplied URL moniker class are asynchronous monikers. *See also* Asynchronous binding, Moniker, and URL moniker.

Automation

A way to manipulate an application's objects from outside the application. Automation is typically used to create applications that expose objects to programming tools and macro languages, create and manipulate one application's objects from another applications, or to create tools for accessing and manipulating objects.

B

Bind context

A COM object that implements the **IBindCtx** interface. Bind contexts are used in moniker operations to hold references to the objects activated when a moniker is bound. The bind context contains parameters that apply to all operations during the binding of a generic composite moniker and provides the moniker implementation with access to information about its environment. *See also* Binding, Generic composite moniker, and Moniker.

Binding

Associating a name with its referent. Specifically, locating the object named by a moniker, putting it into its running state if it isn't already, and returning an interface pointer to it. Objects can be bound at run time (also called *late binding* or *dynamic binding*) or at compile time (also called *static binding*). *See also* Moniker and Running state.

C

Cache
A (usually temporary) local store of information. In OLE, a cache contains information that defines the presentation of a linked or embedded object when the container is opened. *See also* Container, Embedded object, and Linked object.

Cache initialization
Filling a linked or embedded object's cache with presentation data. The **IOleCache** interface provides methods that a container can call to control the data that gets cached for linked or embedded objects. *See also* Container, Embedded object, and Linked object.

Class
The definition of an object in code. In C++, the class of an object is defined as a data type, but this is not the case in other languages. Because OLE can be coded in any language, class is used to refer to the general object definition. *See also* Class factory.

Class factory
A COM object that implements the **IClassFactory** interface and that creates one or more instances of an object identified by a given class identifier(CLSID). *See also* Class identifier.

Class identifier (CLSID)
A globally unique identifier (GUID) associated with an OLE class object. If a class object will be used to create more than one instance of an object, the associated server application should register its CLSID in the system registry so that clients can locate and load the executable code associated with the object(s). Every OLE server or container that allows linking to its embedded objects must register a CLSID for each supported object definition. *See also* Class and Class factory.

Class object
In object-oriented programming, an object whose state is shared by all the objects in a class and whose behavior acts on that classwide state data. In COM, class objects are called class factories, and typically have no behavior except to create new instances of the class. *See also* Class factory.

Client
A COM object that requests services from another object. *See also* Container.

Client site
The display site for an embedded or linked object within a compound document. The client site is the principal means by which an object requests services from its container. *See also* Compound document and Container.

CLSID
See Class identifier.

Commit
To persistently save any changes made to a storage or stream object since it was opened or changes were last saved. *See also* Revert.

Component

An object that encapsulates both data and code, and provides a well-specified set of publicly available services.

Component Object Model (COM)
The OLE object-oriented programming model that defines how objects interact within a single process or between processes. In COM, clients have access to an object through interfaces implemented on the object. *See also* Interface.

Composite menu bar
A shared menu bar composed of menu groups from both a container application and an in-place-activated server application. *See also* In-place activation.

Composite moniker
A moniker that consists of two or more monikers that are treated as a unit. A composite moniker can be non-generic, meaning that its component monikers have special knowledge of each other, or generic, meaning that its component monikers know nothing about each other except that they are monikers. *See also* Generic composite moniker.

Compound document
A document that includes linked or embedded objects, as well as its own native data. *See also* Embedded object and Linked object.

Compound File
An OLE-provided Structured Storage implementation that includes the **IStorage**, **IStream**, and **ILockBytes** interfaces. The **Stg*Xxx*** helper functions create and manipulate compound files. *See also* Structured Storage.

COM object
An object that conforms to the OLE Component Object Model (COM). A COM object is an instance of an object definition, which specifies the object's data and one or more implementations of interfaces on the object. Clients interact with a COM object only through its interfaces. *See also* Component Object Model and Interface.

Connectable object
A COM object that implements, at a minimum, the **IConnectionPointContainer** interface, for the management of connection point objects. Connectable objects support communication from the server to the client. A connectable object creates and manages one or more connection point subobjects, which receive events from interfaces implemented on other objects and send them on to the client. *See also* Connection point object and Advisory sink.

Connection point object
A COM object that is managed by a connectable object and that implements the **IConnectionPoint** interface. One or more connection point objects can be created and managed by a connectable object. Each connection point object manages incoming events from a specific interface on another object and sends those events on to the client. *See also* Connectable object, Advisory sink.

Container
See Container application.

Container application
An application that supports compound documents. The container application provides storage for an embedded or linked object, a site for its display, access to the display site, and an advisory sink for receiving notifications of changes in the object. *See also* Advisory sink, Compound document, Client site, Embedded object, and Linked object.

Containment
A composition technique for implementing COM objects. It allows one object to reuse some or all of the interface implementations of one or more other objects. The outer object acts as a client to the other objects, delegating implementation when it wishes to use the services of one of the contained objects. *Contrast with* Aggregation.

Control
An embeddable, reusable COM object that supports, at a minimum, the **IOleControl** interface. Controls are typically associated with the user interface. They also support communication with a container and can be reused by multiple clients, depending upon licensing criteria. *See also* Licensing.

Control container
An application that supports embedding of controls by implementing the **IOleControlSite** interface. *See also* Control.

Control property
A run-time property that is exposed and managed by the control itself. For example, the font and text size used by the control are control properties. *See also* Run-time property.

Controlling object
The object within an aggregate object that controls which interfaces within the aggregate object are exposed and which are private. The **IUnknown** interface of the controlling object is called the controlling **IUnknown**. Calls to **IUnknown** methods of other objects in the aggregate must be passed to the controlling **IUnknown**. *See also* Aggregate object.

Control site
A structure implemented by a control container for managing the display and storage of a control. Within a given container, each control has a corresponding control site. *See also* Control and Control container.

D

Data transfer object
An object that implements the **IDataObject** interface and contains data to be transferred from one object to another through either the Clipboard or drag-and-drop operations.

Default object handler
A DLL provided with OLE that acts as a surrogate in the processing space of the container application for the real object.

With the default object handler, it is possible to look at an object's stored data without actually activating the object. The default object handler performs other tasks, such as rendering an object

from its cached state when the object is loaded into memory.

Dependent object
A COM object that is typically initialized by another object (the *host* object). Although the dependent object's lifetime may only make sense during the lifetime of the host object, the host object does not function as the controlling **IUnknown** for the dependent object. In contrast, an object is an *aggregated* object when its lifetime (by means of its reference count) is completely controlled by the managing object. *See also* Host object. *Contrast with* Aggregation and Containment.

Direct access mode
One of two access modes in which a storage object can be opened. In direct mode, all changes are immediately committed to the root storage object. *See also* Transacted access mode.

Document object
An OLE document that can display one or more in-place activated views of its data within a native or foreign frame, such as a browser, while retaining full control over its user interface. In addition to implementing the usual OLE document and in-place activation interfaces, a document object must implement **IOleDocument**, and each of its views (in the form of a document view object) must implement **IOleDocumentView**. *See also* Document view, Document view object, and Frame.

Document object container
A container application capable of displaying one or more views of one or more document objects and of managing all contained document objects within a file. Each document object is associated with a document site, and each document site contains one or more document view sites corresponding to the views supported by the document object. A document object container also includes a container frame, which handles menu and toolbar negotiation and the enumeration of contained objects. *See also* Document object, Document site, Document view, Document view site, and Frame.

Document object server
A server application capable of providing document objects. *See also* Document object and Document object container.

Document site
A client site implemented by a document object container for managing the display and storage of a document object. Each document object in a container has a corresponding document site. See also Document object and Document object container.

Document site object
A COM object that implements the **IOleDocumentSite** interface, in addition to the usual client-site interfaces (such as **IOleClientSite**). *See also* Document site.

Document view
A particular presentation of a document's data. A single document object can have one or more views, but a single document view can belong to one and only one document object. *See also* Document object.

Document view object

A COM object that implements the **IOleDocumentView** interface and corresponds to a particular document view. An object with multiple document views aggregates a separate document view object for each view. *See also* Document view.

Document view site

An object aggregated by a document site object for managing the display space for a particular view of a document object. Within a given document site, each document view has a corresponding document view site. *See also* Document object, Document site object, and Document view.

Document view site object

A COM object that is aggregated in a document site object and implements the **IOleInPlaceSite** interface and, optionally, the **IContinueCallback** interface. *See also* Document site object.

Drag and drop

An operation in which the end user uses the mouse or other pointing device to move data to another location in the same window or another window.

E

Embed

To insert an object into a compound document in such a way as to preserve the data formats native to that object, and to enable it to be edited from within its container using tools exposed by its server.

Embedded object

An object whose data is stored in a compound document, but the object runs in the process space of its server. The default object handler provides a surrogate in the processing space of the container application for the real object. *See also* Default object handler, Compound document, and Container application.

.EXE server

See Out-of-process server.

Extended property

A run-time property, such as a control's position and size, that a user would assume to be exposed by the control but is exposed and managed by the container. *See also* Run-time property.

F

File moniker

A moniker based on a path in the file system. File monikers can be used to identify objects that are saved in their own files. A file moniker is a COM object that supports the system-provided implementation of the **IMoniker** interface for the file moniker class. *See also* Item moniker, Generic composite moniker, and Moniker.

Font object

A COM object that provides access to Graphics Device Interface (GDI) fonts by implementing the **IFont** interface.

Format identifier
A GUID that identifies a persistent property set. Also referred to as FMTID. *See also* Property set.

Frame
The part of a container application responsible for negotiating menus, accelerator keys, toolbars, and other shared user-interface elements with an embedded COM object or a document object. *See also* Document object and Embedded object.

Frame object
A COM object that implements the **IOleInPlaceFrame** interface and, optionally, the **IOleCommandTarget** interface.

G

Generic composite moniker
A sequenced collection of monikers, starting with a file moniker to provide the document-level path and continuing with one or more item monikers that, taken as a whole, uniquely identifies an object. *See also* Composite moniker, Item moniker, and File moniker.

H

Helper function
A function that encapsulates calls to other functions and interface methods publicly available in the OLE SDK. Helper functions are a convenient way to call frequently used sequences of function and method calls that accomplish common tasks.

Host object
A COM object that forms a hierarchical relationship with one or more other COM objects, known as the *dependent* objects. Typically, the host object instantiates the dependent objects, and their existence only makes sense within the lifetime of the host object. However, the host object does not act as the controlling **IUnknown** for the dependent objects, nor does it directly delegate to the interface implementations of those objects. *See also* Dependent object.

HRESULT
An opaque result handle defined to be zero for a successful return from a function and nonzero if error or status information is returned. To convert an HRESULT into the more detailed SCODE, applications call **GetScode**(). *See also* SCODE.

Hyperlink object
A COM object that implements, at a minimum, the **IHlink** interface and acts as a link to an object at another location (the target). A hyperlink is made up of four parts: a moniker that identifies the target's location; a string for the location within the target; a friendly, or displayable, name for the target; and a string that can contain additional parameters.

Hyperlink browse context
A COM object that implements the **IHlinkBrowseContext** interface and maintains the hyperlink navigation stack. The browse context object manages the hyperlink frame window and hyperlink target object's window. *See also* Hyperlink target.

Hyperlink container
A container application that supports hyperlinks by implementing the **IHlinkSite** interface and, if the container's objects can be targets of other hyperlinks, the **IHlinkTarget** interface. *See also* Container.

Hyperlink frame object
A COM object that implements the **IHlinkFrame** interface and controls the top-level navigation and display of hyperlinks for the frame's container and the hyperlink target's server.

Hyperlink site object
A COM object that implements the **IHlinkSite** interface and supplies either the moniker or interface identifier of its hyperlink container. One hyperlink site can serve multiple hyperlinks. *See also* Hyperlink and Hyperlink container.

Hyperlink target object
A COM object that implements the **IHlinkTarget** interface and supplies its moniker, friendly name, and other information that other hyperlink objects will use to navigate to it.

I

In parameter
A parameter that is allocated, set, and freed by the caller of a function or interface method. An In parameter is not modified by the called function. *See also* In/Out parameter and Out parameter.

In/Out parameter
A parameter that is initially allocated by the caller of a function or interface method, and set, freed, and reallocated, if necessary, by the process that is called. *See also* In parameter and Out parameter.

In-place activation
Editing an embedded object within the window of its container, using tools provided by the server. Linked objects do not support in-place activation; they are always edited in the window of the server. *See also* Embedded object and Linked object.

In-process server
A server implemented as a DLL that runs in the process space of the client. *See also* Out-of-process server, Local server, and Remote server.

Instance
An object for which memory is allocated or which is persistent.

Interface
A group of semantically related functions that provide access to a COM object. Each OLE interface defines a contract that allows objects to interact according to the Component Object Model (COM). While OLE provides many interface implementations, most interfaces can also be implemented by developers designing OLE applications. *See also* Component Object Model and COM object.

Interface identifier (IID)

A globally unique identifier (GUID) associated with an interface. Some functions take IIDs as parameters to allow the caller to specify which interface pointer should be returned.

Item moniker

A moniker based on a string that identifies an object in a container. Item monikers can identify objects smaller than a file, including embedded objects in a compound document, or a pseudo-object (like a range of cells in a spreadsheet). *See also* File moniker, Generic composite moniker, Moniker, and Pseudo-object.

L

Licensing

A feature of COM that provides control over object creation. Licensed objects can be created only by clients that are authorized to use them. Licensing is implemented in COM through the **IClassFactory2** interface and by support for a license key that can be passed at run time.

Link object

A COM object that is created when a linked COM object is created or loaded. The link object is provided by OLE and implements the **IOleLink** interface.

Linked object

A COM object whose source data physically resides where it was initially created. Only a moniker that represents the source data and the appropriate presentation data are kept with the compound document. Changes made to the link source are automatically reflected in the linked object. *See also* Link source.

Link source

The data that is the source of a linked object. A link source may be a file or a portion of a file, such as a selected range of cells within a file (also called a pseudo object). *See also* Linked object.

Loaded state

The state of an object after its data structures have been loaded into memory and are accessible to the client process. *See also* Active state, Passive state, and Running state.

Local server

An out-of-process server implemented as an .EXE application running on the same machine as its client application. *See also* In-process server, Out-of-process server, and Remote server.

Lock

A pointer held to—and possibly, a reference count incremented on—a running object. OLE defines two types of locks that can be held on an object: *strong* and *weak*. To implement a strong lock, a server must maintain both a pointer and a reference count, so that the object will remain "locked" in memory at least until the server calls **Release**. To implement a weak lock, the server maintains only a pointer to the object, so that the object can be destroyed by another process.

M

Marshaling

Packaging and sending interface method calls across thread or process boundaries.

Media type
An extension of MIME that allows data format negotiation between a client and an object. *See also* Multipurpose Internet Mail Extension (MIME).

MIME
See Multipurpose Internet Mail Extension.

MIME content type
See Media type.

Multipurpose Internet Mail Extension (MIME)
An Internet protocol originally developed to allow exchange of electronic mail messages with rich content across heterogeneous network, machine, and e-mail environments. In practice, MIME has also been adopted and extended by non-mail applications.

Moniker
An object that implements the **IMoniker** interface. A moniker acts as a name that uniquely identifies a COM object. In the same way that a path identifies a file in the file system, a moniker identifies a COM object in the directory namespace. *See also* Binding.

Moniker class
An implementation of the **IMoniker** interface. System-supplied moniker classes include file monikers, item monikers, generic composite monikers, anti-monikers, pointer monikers, and URL monikers.

Moniker client
An application that uses monikers to acquire interface pointers to objects managed by another application.

Moniker provider
An application that makes available monikers that identify the objects it manages, so that the objects are accessible to other applications.

N

Namespace extension
An in-process COM object that implements **IShellFolder**, **IPersistFolder**, and **IShellView**, which are sometimes referred to as the namespace extension interfaces. A namespace extension is used either to extend the shell's namespace or to create a separate namespace. Primary users are the Windows Explorer and common file dialog boxes.

Native data
The data used by an OLE server application when editing an embedded object. *See also* Presentation data.

O

Object
In OLE, a programming structure encapsulating both data and functionality that are defined and allocated as a single unit and for which the only public access is through the programming structure's interfaces. A COM object must support, at a minimum, the **IUnknown** interface, which maintains the object's existence while it is being used and provides access to the object's other interfaces. *See also* COM and Interface.

Object handler
See Default object handler.

Object state
The relationship between a compound document object in its container and the application responsible for the object's creation: *active, passive*, *loaded*, or *running*. Passive objects are stored on disk or in a database, and the object is not selected or active. In the loaded state, the object's data structures have been loaded into memory, but they are not available for operations such as editing. Running objects are both loaded and available for all operations. Active objects are running objects that have a visible user interface.

Object type name
A unique identification string that is stored as part of the information available for an object in the registration database.

OLE
Microsoft's object-based technology for sharing information and services across process and machine boundaries.

OLE Automation
See Automation.

OLE control
See Control.

Out-of-process server
A server, implemented as an .EXE application, which runs outside the process of its client, either on the same machine or a remote machine. *See also* Local server and Remote server.

Out parameter
A parameter that is allocated and freed by the caller, but its value is set by the function being called. *See also* In parameter and In/Out parameter.

P

Passive state
The state of a COM object when it is stored (on disk or in a database). The object is not selected or active. *See also* Active state, Loaded state, Object state, and Running state.

Persistent properties
Information that can be stored persistently as part of a storage object such as a file or directory. Persistent properties are grouped into property sets, which can be displayed and edited.

Persistent properties are different from the run-time properties of objects created with OLE Controls and Automation technologies, which can be used to affect system behavior. The PROPVARIANT structure defines all valid types of persistent properties, whereas the VARIANT structure defines all valid types of run-time properties. *See also* Compound files, Property, and Property sets.

Persistent storage
Storage of a file or object in a medium such as a file system or database so that the object and its data persist when the file is closed and then re-opened at a later time.

Picture object
A COM object that provides access to GDI images by implementing the **IPicture** interface.

Pointer moniker
A moniker that maps an interface pointer to an object in memory. Whereas most monikers identify objects that can be persistently stored, pointer monikers identify objects that cannot. They allow such objects to participate in a moniker binding operation.

Presentation data
The data used by a container to display embedded or linked objects. *See also* Native data.

Primary verb
The action associated with the most common or preferred operation users perform on an object. The primary verb is always defined as verb zero in the system registration database. An object's primary verb is executed by double-clicking on the object.

Property
Information that is associated with an object. In OLE, properties fall into two categories: run-time properties and persistent properties. Run-time properties are typically associated with control objects or their containers. For example, background color is a run-time property set by a control's container. Persistent properties are associated with stored objects. *See also* Persistent properties and Run-time properties.

Property frame
The user interface mechanism that displays one or more property pages for a control. The OLE Controls run-time system provides a standard implementation of a property frame that can be accessed by using the **OleCreatePropertyFrame** helper funtion. *See also* Control and Property page.

Property identifier
A four-byte signed integer that identifies a persistent property within a property set. *See also* Persistent property and Property set.

Property page
A COM object with its own CLSID that is part of a user interface, implemented by a control, and

allows the control's properties to be viewed and set. Property page objects implement the **IPropertyPage** interface. *See also* CLSID, Control.

Property page site

The location within a property frame where a property page is displayed. The property frame implements the **IPropertyPageSite** interface, which contains methods to manage the sites of each of the property pages supplied by a control. *See also* Property frame.

Property set

A logically related group of properties that is associated with a persistently stored object. To create, open, delete, or enumerate one or more property sets, implement the **IPropertySetStorage** interface. If you are using compound files, you can use OLE's implementation of this interface rather than implementing your own. *See also* Persistent properties.

Property set storage

A COM storage object that holds a property set. A property set storage is a dependent object associated with and managed by a storage object. *See also* Dependent object, Property set.

Property sheet

A set of property pages for one or more objects. *See also* Property page.

Proxy

An interface-specific object that packages parameters for that interface in preparation for a remote method call. A proxy runs in the address space of the sender and communicates with a corresponding stub in the receiver's address space. *See also* Stub, Marshaling, and Unmarshaling.

Proxy manager

In standard marshaling, a proxy that manages all the interface proxies for a single object. *See also* Marshaling, Proxy.

Pseudo-object

A portion of a document or embedded object, such as a range of cells in a spreadsheet, that can be the source for a COM object.

R

Reference counting

Keeping a count of each interface pointer held on an object to ensure that the object is not destroyed before all references to it are released. *See also* Lock.

Relative moniker

A moniker that specifies the location of an object relative to the location of another object. A relative moniker is analogous to a relative path, such as ..\backup\report.old. *See also* Moniker.

Remote Server

A server application, implemented as an EXE, running on a different machine from the client application using it. *See also* In-process server, Local server, and Out-of-process server.

Revert
To discard any changes made to an object since the last time the changes were committed or the object's storage was opened. *See also* Commit and Transacted access mode.

Root storage object
The outermost storage object in a document. A root storage object can contain other nested storage and stream objects. For example, a compound document is saved on disk as a series of storage and stream objects within a root storage object. *See also* Compound document, Storage object, and Stream object.

Running state
The state of a COM object when its server application is running and it is possible to access its interfaces and receive notification of changes. *See also* Active state, Loaded state, Passive state.

Running Object Table (ROT)
A globally accessible table on each computer that keeps track of all COM objects in the running state that can be identified by a moniker. Moniker providers register an object in the table, which increments the object's reference count. Before the object can be destroyed, its moniker must be released from the table. *See also* Running state.

Run-time property
Discrete state information associated with a control object or its container. There are three types of run-time properties: ambient properties, control properties, and extended properties. *See also* Ambient property, Control property, and Extended property. *Contrast with* Persistent property.

S

SCODE
A DWORD value that is used to return detailed information to the caller of an interface method or function. *See also* HRESULT.

Self-registration
The process by which a server can perform its own registry operations.

Server application
An application that can create COM objects. Container applications can then embed or link to these objects. *See also* Container application.

Sink
See Advisory sink.

State
See Active state, Loaded state, Object state, Passive state, and Running state.

Static object
An object that contains only a presentation, with no native data. A container can treat a static object as though it were a linked or embedded object, except that it is not possible to edit a static object.

A static object can result, for example, from the breaking of a link on a linked object—that is, the server application is unavailable, or the user doesn't want the linked object to be updated anymore. *See also* Native data.

Storage object
A COM object that implements the **IStorage** interface. A storage object contains nested storage objects or stream objects, resulting in the equivalent of a directory/file structure within a single file. *See also* Root storage object and Stream object.

Stream object
A COM object that implements the **IStream** interface. A stream object is analogous to a file in a directory/file system. *See also* Storage object.

Strong lock
See Lock.

Structured Storage
OLE's technology for storing compound files in native file systems. *See also* Compound file, Storage object, and Stream object.

Stub
When a function's or interface method's parameters are marshaled across a process boundary, the stub is an interface-specific object that unpackages the marshaled parameters and calls the required method. The stub runs in the receiver's address space and communicates with a corresponding proxy in the sender's address space. *See also* Proxy, Marshaling, and Unmarshaling.

Stub manager
Manages all of the interface stubs for a single object.

Subobject
See Dependent object.

Synchronous call
A function call that does not allow further instructions in the calling process to be executed until the function returns. *See also* Asychronous call.

System registry
A system-wide repository of information supported by Windows, which contains information about the system and its applications, including OLE clients and servers.

T

Transacted access mode
One of two access modes in which a storage object can be opened. When opened in transacted mode, changes are stored in buffers until the root storage object commits its changes. *See also* Direct access mode, Commit, Revert, and Root storage object.

Type information
Information about an object's class provided by a type library. To provide type information, a COM object implements the **IProvideClassInfo** interface.

U

Uniform data transfer
A model for transferring data via the Clipboard, drag and drop, or Automation. Objects conforming to this model implement the **IDataObject** interface. This model replaces DDE (dynamic data exchange). *See also* Data transfer object.

Unmarshaling
Unpacking parameters that have been sent to a proxy across process boundaries.

Universal resource locator (URL)
The identifier used by the World Wide Web for the names and locations of objects on the Internet. OLE provides a moniker class, URL moniker, whose implementation can be used to bind a client to objects identified by a URL. *See also* URL moniker.

URL moniker
A moniker based on a universal resource locator (URL). A client can use URL monikers to bind to objects that reside on the Internet. The system-supplied URL moniker class supports both synchronous and asynchronous binding. *See also* Asynchronous binding.

V

Virtual Table (VTBL)
An array of pointers to interface method implementations. *See also* Interface.

Visual Editing
A term in end-user documents that refers to the user's ability to interact with a compound-document object in the context of its container. The term most often used by developers is in-place activation.

W

Weak lock
See Lock .

Legal Information

Automation Programmer's Reference

Introduction

This book provides procedural and reference information for Automation (formerly called OLE Automation). While Automation runs on other platforms, the focus of this document is applications that use the Microsoft® Windows® 32-bit operating system.

To get the most out of this book, you should be familiar with:

- The C++ and Microsoft Visual Basic® programming languages.
- The Microsoft Windows 95® and Windows NT® programming environments.
- The OLE protocols are implemented through dynamic link libraries (DLLs) that are used in conjunction with other Microsoft Windows programs.
- The Component Object Model (COM).

This information is also available in the Win32® Software Development Kit (SDK), which is contained on the Microsoft Developer Network (MSDN).

About This Book

This book contains the following chapters and appendixes:

Chapter 1, "Overview of Automation," introduces the basic concepts of Automation and identifies the components.

Chapter 2, "Exposing ActiveX Objects," shows how to write and expose programmable objects for use by ActiveX clients, and demonstrates programming techniques with sample code.

Chapter 3, "Accessing ActiveX Objects," explains how to write applications and programming tools that access exposed objects.

Chapter 4, "Standard Objects and Naming Guidelines," lists the standard ActiveX objects that are recommended for most applications, and describes naming conventions for objects.

Chapter 5, "Dispatch Interface and API Functions," describes the interfaces and functions that support access to exposed objects.

Chapter 6, "Data Types, Structures, and Enumerations," describes functions that manipulate arrays, strings, and variant types of data within Automation.

Chapter 7, "Conversion and Manipulation Functions," describes Automation API functions.

Chapter 8, "Type Libraries and the Object Description Language," describes the Microsoft Interface Definition Language (MIDL) compiler and the MkTypLib tool and its source file language. MIDL and MkTypLib creates type libraries according to the descriptions you provide.

Chapter 9, "Type Description Interfaces," describes the interfaces and functions that allow programs to read and bind to the descriptions of objects in a type library.

Chapter 10, "Type Building Interfaces," describes interfaces and functions that build type libraries.

Chapter 11, "Error Handling Interfaces," describes how Automation error handling interfaces define and return error information.

Appendix A, "National Language Support Functions," describes functions for 32-bit and 16-bit systems that support multiple national languages.

Appendix B, "File Requirements," lists the files you and your customers need to run Automation applications.

Appendix C, "Information for Visual Basic Programmers," lists the Automation APIs that are called by Visual Basic statements.

Appendix D, "String Comparisons," describes the string comparison rules applied by Automation.

Appendix E, "Managing GUIDs," provides supplemental information on globally unique identifiers.

The **Glossary** defines some of the terms that are useful in understanding Automation.

Other Sources of Information

Automation and ActiveX technologies are implementations of the OLE COM, which provides mechanisms for in-place activation, structured file storage, and many other application features. These parts of OLE are fully described in the following sources:

- *OLE Programmer's Guide and Reference* describes the COM, in-place activation, visual editing, structured file storage, and application registration in terms of the APIs and interfaces provided by OLE.
- *Microsoft Interface Definition Language Programmer's Guide and Reference*, contained in the Win32 SDK on the MSDN, describes the MIDL compiler.
- *Inside OLE, Second Edition,* by Kraig Brockschmidt, published by Microsoft Press, provides introductory and how-to information about implementing OLE objects and containers.
- If you are developing C++ applications, the *ActiveX Control Developer's Kit* and *Microsoft Visual C++ version 4.x* product documentation describes how to develop Automation applications using C++.

For technical support, see the *OLE Programmer's Guide and Reference* and the documentation for the product with which you received OLE. Support for Automation is also provided on the World Wide Web on the Microsoft home page at www.microsoft.com.

Document Conventions

The following typographical conventions are used throughout this book.

Convention	Meaning
bold	Indicates a word that is a function name, method, attribute, or other fixed part of a programming language, the Microsoft Windows operating system, or the Application Programming Interface (API). For example, **DispInvoke** is an OLE-specific function. These words must always be typed exactly as they are printed.
italic	Indicates a word that is a placeholder or variable. For example, *ClassName* would be a placeholder for any ActiveX object class name. Function parameters in API reference material are italic to indicate that any variable name can be used. In addition, ActiveX and OLE terms are italicized at first use to highlight their definition.
UPPERCASE	Indicates a constant or data structures. For example, E_INVALIDARG is a constant.
InitialCaps	Indicates the name of an object, property, method, event, or file name. For example, the Application object.
`monospace`	Indicates source code and syntax spacing. For example: `*pdwRegisterCF = 0;`

Note The interface syntax in this book follows the variable-naming convention known as Hungarian notation, invented by programmer Charles Simonyi. Variables are prefixed with lowercase letters indicating their data type. For example, *lpszNewDocname* would be a long pointer to a zero-terminated string named *NewDocname*. For more information about Hungarian notation, refer to *Programming Windows* by Charles Petzold.

Overview of Automation

Automation (formerly called OLE Automation) is a technology that allows software packages to expose their unique features to scripting tools and other applications. Automation uses the Component Object Model (COM), but may be implemented independently from other OLE features, such as in-place activation. Using Automation, you can:

- Create applications and programming tools that expose objects.
- Create and manipulate objects exposed in one application from another application.
- Create tools that access and manipulate objects. These tools can include embedded macro languages, external programming tools, object browsers, and compilers.

The objects an application or programming tool exposes are called *ActiveX™ objects*. Applications and programming tools that access those objects are called *ActiveX clients*. ActiveX objects and clients interact as follows:

{ewc msdncd, EWGraphic, bsd23523 0 /a "SDK_01.WMF"}

Applications and other software packages that support ActiveX technology define and expose objects which can be acted on by *ActiveX components*. ActiveX components are physical files (for example .exe and .dll files) that contain classes, which are definitions of objects. Type information describes the exposed objects, and can be used by ActiveX components at either compile time or at run time.

Why Expose Objects?

Exposing objects provides a way to manipulate an application's tools programmatically. This allows customers to use a programming tool that automates repetitive tasks that might not have been anticipated.

For example, Microsoft Excel® exposes a variety of objects that can be used to build applications. One such object is the Workbook, which contains a group of related worksheets, charts, and macros – the Microsoft Excel equivalent of a three-ring binder. Using Automation, you could write an application that accesses Microsoft Excel Workbook objects, possibly to print them, as in the following diagram:

{ewc msdncd, EWGraphic, bsd23523 1 /a "SDK_02.BMP"}

With Automation, solution providers can use your general-purpose objects to build applications that target a specific task. For example, you could use a general-purpose drawing tool to expose objects that draw boxes, lines, and arrows, insert text, and so forth. Another programmer could build a flowchart tool by accessing the exposed objects and then adding a user interface and other application-specific features.

Exposing objects to Automation or supporting Automation within a macro language offers several benefits.

- Exposed objects from many applications are available in a single programming environment. Software developers can choose from these objects to create solutions that span applications.
- Exposed objects are accessible from any macro language or programming tool that implements Automation. Systems integrators are not limited to the programming language in which the objects were developed. Instead, they can choose the programming tool or macro language that best suits their own needs and capabilities.
- Object names can remain consistent across versions of an application, and can conform automatically to the user's national language.

What Is An ActiveX Object?

An ActiveX object is an instance of a class that exposes properties, methods, and events to ActiveX clients. ActiveX objects support the COM. An *ActiveX component* is an application or library that is capable of creating one or more ActiveX objects. For example, Microsoft Excel exposes many objects that you can use to create new applications and programming tools. Within Microsoft Excel, objects are organized hierarchically, with an object named Application at the top of the hierarchy.

The following figure shows some of the objects in Microsoft Excel.

{ewc msdncd, EWGraphic, bsd23523 2 /a "SDK_03.WMF"}

Each ActiveX object has its own unique member functions. When the member functions are exposed, it makes the object programmable by ActiveX clients. Three types of members for an object can be exposed:

- *Methods* are actions that an object can perform. For example, the Worksheet object in Microsoft Excel provides a **Calculate** method that recalculates the values in the worksheet.
- *Properties* are functions that access information about the state of an object. The Worksheet object's **Visible** property determines whether the worksheet is visible.
- *Events* are actions recognized by an object, such as clicking the mouse or pressing a key. You can write code to respond to such actions. In Automation, an event is a method that is called, rather than implemented, by an object.

For example, you might expose the following objects in a document-based application by implementing these methods and properties:

ActiveX object	Methods	Properties
Application	**Help** **Quit** **Save** **Repeat** **Undo**	**ActiveDocument** **Application** **Caption** **DefaultFilePath** **Documents** **Height** **Name** **Parent** **Path** **Printers** **StatusBar** **Top** **Value** **Visible** **Width**
Document	**Activate** **Close** **NewWindow** **Print** **PrintPreview** **RevertToSaved** **Save** **SaveAs**	**Application** **Author** **Comments** **FullName** **Keywords** **Name** **Parent** **Path** **ReadOnly** **Saved** **Subject** **Title**

Value

Often, an application works with several instances of an object which together make up a *collection object*. For example, an ActiveX application based on Microsoft Excel might have multiple workbooks. To provide an easy way to access and program the workbooks, Microsoft Excel exposes an object named Workbooks, which refers to all of the current Workbook objects. Workbooks is a collection object.

In the preceding figure, collection objects in Microsoft Excel are shaded. Collection objects let you work iteratively with the objects they manage. If an application is created with a multiple-document interface (MDI), it might expose a collection object named Documents with the methods and properties in the following table.

Collection object	Methods	Properties
Documents	**Add**	**Application**
	Close	**Count**
	Item	**Parent**
	Open	

What Is An ActiveX Client?

An ActiveX client is an application or programming tool that manipulates one or more ActiveX objects. The objects can exist in the same application or in another application. Clients can use existing objects, create new instances of objects, get and set properties, and invoke methods supported by the object.

Microsoft Visual Basic® is an ActiveX client. You can use Visual Basic and similar programming tools to create packaged scripts that access Automation objects. You can also create clients by doing the following:

- Writing code within an application that accesses another application's exposed objects through Automation.
- Revising an existing programming tool, such as an embedded macro language, to add support for Automation.
- Developing a new application, such as a compiler or type information browser, that supports Automation.

How Do Clients and Objects Interact?

ActiveX clients can access objects in two different ways:

- By using the **IDispatch** *interface*.
- By calling one of the member functions directly in the object's *virtual function table* (VTBL).

An Automation interface is a group of related functions that provide a service. All ActiveX objects must implement the **IUnknown** interface because it manages all of the other interfaces that are supported by the object. The **IDispatch** interface, which derives from the **IUnknown** interface, consists of functions that allow access to the methods and properties of ActiveX objects.

A *custom interface* is a COM interface that is not defined as part of OLE. Any user-defined interface is a custom interface.

The VTBL lists the addresses of all the properties and methods that are members of an object, including the member functions of the interfaces that it supports. The first three members of the VTBL are the members of the **IUnknown** interface. Subsequent entries are members of the other supported interfaces.

The following figure shows the VTBL for an object that supports the **IUnknown** and **IDispatch** interfaces.

{ewc msdncd, EWGraphic, bsd23523 3 /a "SDK_04.WMF"}

If an object does not support **IDispatch**, the member entries of the object's custom interfaces immediately follow the members of **IUnknown**. For example, the following figure shows the VTBL for an object that supports a custom interface named **IMyInterface**.

{ewc msdncd, EWGraphic, bsd23523 4 /a "SDK_05.WMF"}

When an object for Automation is exposed, you must decide whether to implement an **IDispatch** interface, a VTBL interface, or both. Microsoft strongly recommends that objects provide a *dual interface*, which supports both access methods.

In a dual interface, the first three entries in the VTBL are the members of **IUnknown**, the next four entries are the members of **IDispatch**, and the subsequent entries are the addresses of the members of the dual interface.

The following figure shows the VTBL for an object that supports a dual interface named **IMyInterface**:

{ewc msdncd, EWGraphic, bsd23523 5 /a "SDK_06.WMF"}

In addition to providing access to objects, Automation also provides information about exposed objects. By using **IDispatch** or a *type library*, an ActiveX client or programming tool can determine which interfaces an object supports, as well as the names of its members. Type libraries, which are files or parts of files that describe the type of one or more ActiveX objects, are especially useful because they can be accessed at compile time. For information on type libraries, refer to "What Is a Type Library?" later in this chapter, and "Type Libraries" in Chapter 2, "Exposing ActiveX Objects."

Accessing an Object Through the IDispatch Interface

ActiveX clients can use the **IDispatch** interface to access objects that implement the interface. The client must first create the object, and then query the object's **IUnknown** interface for a pointer to its **IDispatch** interface.

Although programmers might know objects, methods, and properties by name, **IDispatch** keeps track of them internally with a number called the *dispatch identifier* (DISPID). Before an ActiveX client can access a property or method, it must have the DISPID that maps to the name of the member.

With the DISPID, a client can call the member **IDispatch::Invoke** to access the property or invoke the method, and then package the parameters for the property or method into one of the **IDispatch::Invoke** parameters.

The object's implementation of **IDispatch::Invoke** must then unpackage the parameters, call the property or method, and handle any errors that occur. When the property or method returns, the object passes its return value back to the client through an **IDispatch::Invoke** parameter.

DISPIDs are available at run time, and, in some circumstances, at compile time. At run time, clients get DISPIDs by calling the **IDispatch::GetIDsOfNames** function. This is called *late binding* because the controller binds to the property or method at run time instead of compile time.

The DISPID of each property or method is fixed, and is part of the object's type description. If the object is described in a type library, an ActiveX client can read the DISPIDs from the type library at compile time, and avoid calling **IDispatch::GetIDsOfNames**. This is called *ID binding*. Because it requires only one call to **IDispatch** (the call to **Invoke**), rather than the two calls required by late binding, it is generally about twice as fast. Late-binding clients can improve performance by caching DISPIDs after retrieving them, so that **IDispatch::GetIDsOfNames** is called only once for each property or method.

Accessing an Object Through the VTBL

Automation allows an ActiveX client to call a method or property accessor function directly, either within or across processes. This approach, called *VTBL binding*, does not use the **IDispatch** interface. The client obtains type information from the type library at compile time, and then calls the methods and functions directly. VTBL binding is faster than both ID binding and late binding because the access is direct, and no calls are made through **IDispatch**.

In-Process and Out-of-Process Server Objects

ActiveX objects can exist in the same process as their controller, or in a different process. *In-process server* objects are implemented in a dynamic-link library (DLL) and are run in the process space of the controller. Because they are contained in a DLL, they cannot be run as stand-alone objects. *Out-of-process server* objects are implemented in an executable file and are run in a separate process space. Access to in-process objects is much faster than to out-of-process server objects because Automation does not need to make remote procedure calls across the process boundary.

The access mechanism (**IDispatch** or VTBL) and the location of an object (in-process or out-of-process server) determine the fixed overhead required for access. The most important factor in performance, however, is the quantity and nature of the work performed by the methods and procedures that are invoked. If a method is time consuming or requires remote procedure calls, the overhead of the call to **IDispatch** may make a call to VTBL more appropriate.

What Is a Type Library?

A type library is a file or part of a file that describes the type of one or more ActiveX objects. Type libraries do not store objects; they store type information. By accessing a type library, applications and browsers can determine the characteristics of an object, such as the interfaces supported by the object and the names and addresses of the members of each interface. A member can also be invoked through a type library. For details about the interfaces, refer to Chapter 9, "Type Description Interfaces."

When ActiveX objects are exposed, you should create a type library to make objects easily accessible to other developers. The simplest way to do this is to describe objects in an Object Description Language (.odl) file, and then compile the file with the MkTypLib tool, as described in Chapter 8, "Type Libraries and the Object Description Language."

For this release of Automation, the Microsoft Interface Definition Language (MIDL) compiler can be used to generate a type library. For information about the MIDL compiler, refer to the *Microsoft Interface Definition Language Programmer's Guide and Reference* in the Win32 Software Development Kit (SDK) section of the Microsoft Developer's Network (MSDN).

Exposing ActiveX Objects

Exposing objects makes them available for programmatic use by other applications and programming tools. This chapter discusses how to design an application that exposes objects, and then uses various samples from the *Microsoft OLE Programmer's Guide and Reference* in the Win32 Software Development Kit (SDK) to demonstrate how to implement the design.

Note Throughout this chapter, the file names of sample applications appear in parentheses before the sample code.

Exposing Objects

To expose ActiveX objects, you write code to initialize the objects, implement the objects, and then release OLE when the application terminates.

Quick Info

To initialize exposed objects

1. Initialize OLE.
2. Register the class factories of the exposed objects.
3. Register the active object.

Quick Info

To implement exposed objects

1. Implement the **IUnknown**, **IDispatch**, and virtual function table (VTBL) interfaces for the objects.
2. Implement the properties and methods of the objects.

Quick Info

To release OLE when the application terminates

1. Revoke the registration of the class factories and revoke the active object.
2. Uninitialize OLE.

Quick Info

To retrieve active objects for use by others

1. Create an object description language (.odl) file or create a library section in an interface definition language (.idl) file that describes the properties and methods of the exposed objects. Use MkTypLib to compile the .odl file into a type library or use the Microsoft Interface Definition Language (MIDL) compiler for both the .idl file and .odl file.
2. Create a registration (.reg) file for the application.

Initializing Exposed Objects

To initialize OLE and the exposed objects, use the following functions:

- **OleInitialize**– Initializes OLE.
- **CoRegisterClassObject**–Registers the object's class factory with OLE so other applications can use it to create new objects.
- **RegisterActiveObject**– Registers the active object so other applications can connect to an existing object.

Implementing Exposed Objects

The following figure shows the interfaces that you should implement to expose ActiveX objects.

{ewc msdncd, EWGraphic, bsd23524 0 /a "SDK_01.WMF"}

The member functions are listed under each interface name.

Implementing a Class Factory

Before OLE can create an object, it needs access to the object's *class factory*. The class factory implements the **IClassFactory** interface. For detailed information about this interface, see the *Microsoft OLE Programmer's Guide and Reference* and *Inside OLE, Second Edition,* published by Microsoft Press. This chapter describes only what you must do to expose objects for Automation.

It is important to implement a class factory for objects that may be created explicitly through the OLE function **CoCreateInstance,** or through the Visual Basic **New** keyword. For example, an application can expose an Application object for creation, but may have many other programmable objects that can be created or destroyed by referencing a member of the Application object. In this case, only the Application object would need a class factory.

For each class factory, you need to implement the following two member functions of the **IClassFactory** interface, which provide services for OLE API functions. The prototypes for the member functions reside in the file Ole2.h.

- **CreateInstance** – Creates an instance of the object's class.
- **LockServer** – Prevents the object's server from shutting down, even if the last instance of the object is released. **LockServer** can improve the performance of applications that frequently create and release objects.

In general, the **CreateInstance** method should create a new instance of the object's class. For the Application object, however, the **CreateInstance** method should return the existing instance of the Application object, which is registered in the running object table (ROT).

The class factory object implements the **IClassFactory** and **IUnknown** interfaces. All objects must implement **IUnknown**, which allows ActiveX clients to determine which interfaces the object supports. A class factory can create instances of a class.

The object implements two interfaces: **IUnknown** and **IMyInterface**. The interface **IMyInterface** is a **dual** interface, which supports both late binding through **IDispatch,** and early binding through the VTBL. The **dual** interface provides two ways to invoke the object's methods and properties. **IDispatch** includes the member functions **GetIDsOfNames, GetTypeInfo, GetTypeInfoCount,** and **Invoke**.

Member1 and Member2 are the members of **IMyInterface**. These members are available as direct entry points through the object's VTBL. They can also be accessed through **IDispatch::Invoke**.

You must decide how to handle errors that occur in the exposed objects. If an object supports a **dual** interface and needs to return detailed, contextual error information, you also need to implement the Automation error interface, **IErrorInfo**.

In addition to writing code to implement objects, you must create a type library and a registration file. Describe the types of exposed objects in the library section of the MIDL file or create an .odl file. Use the MIDL compiler or the MkTypLib tool to compile the .odl file. A type library (.tlb) file and a header (.h) file are created. The registration file provides information that the operating system and OLE need to locate objects.

Exposing the Application Object

Any document-based, user-interactive applications that expose ActiveX objects should have one top-level object named the Application object. This object is initialized as the active object when an application starts.

The Application object identifies the application and provides a way for ActiveX clients to bind to and navigate the application's exposed objects. All other exposed objects are subordinate to the Application object; it is the root-level object in the object hierarchy.

The names of the Application object's members are part of the global name space, so ActiveX clients do not need to qualify them. For example, if MyApplication is the name of the Application object, a Visual Basic program can refer to a method of MyApplication as MyApplication.MyMethod or simply MyMethod. However, you should be careful not to overload the Application object with too many members because it can cause ambiguity and decrease performance. A large, complicated application with many members should be organized hierarchically, with a few generalized objects at the top, branching out into smaller, more specialized objects.

The following chart shows how applications should expose their Application and Document objects.

Command line	Multiple-document interface application	Single-document interface application
/Embedding	Expose class factories for document classes, but not for the application. Call **RegisterActiveObject** for the Application object.	Expose class factories for document class, but not for the application. Call **RegisterActiveObject** for the Application object.
/Automation	Expose class factories for document classes. Expose class factory for the application using **RegisterClassObject**. Call **RegisterActiveObject** for the Application object.	Do not expose class factory for document class. Expose class factory for the Application object using **RegisterClassObject**. Call **RegisterActiveObject** for the Application object.
No OLE switches	Expose class factories for document classes, but not for the application. Call **RegisterActiveObject** for the Application object.	Call **RegisterActiveObject** for the Application object.

The call to **RegisterActiveObject** enters the Application object in OLE's running object table (ROT), so ActiveX clients can retrieve the active object instead of creating a new instance. Visual Basic applications can use the **GetObject** statement to access an existing object.

Creating a Registration File

Before an application can use OLE and Automation, the OLE objects must be registered with the user's system registration database. OLE provides sample registration files to perform this task for the OLE objects and the sample applications. Registration makes the following possible:

- ActiveX clients can create instances of the objects through **CoCreateInstance**.
- Automation tools can find the type libraries that are installed on the user's computer.
- OLE can find remoting code for the interfaces.

You can use the **DLLRegisterServer** function to register all objects implemented by a DLL. This function registers the class IDs for each object, the programmatic IDs for each application, and the type library. For details, refer to the description of **DLLRegisterServer** in *Programming with MFC*, provided with Microsoft Visual C++ version 4.1 and later product documentation.

The following sections give a brief overview of the syntax used in registering ActiveX objects. Descriptions of the process are provided later in this chapter. For detailed information, refer to the *Microsoft OLE Programmer's Guide and Reference*.

Registering the Application

Registration maps the *programmatic ID* (ProgID) of the application to a unique class ID (CLSID), so that you can create instances of the application by name, rather than by CLSID. For example, registering Microsoft Excel associates a CLSID with the ProgID `Excel.Application`. In Visual Basic, you use the ProgID to create an instance of the application as follows:

```
SET xl = CreateObject("Excel.Application")
```

By passing the ProgID to **CLSIDFromProgID**, you can get the corresponding CLSID for use in **CoCreateInstance**. Only applications that will be used in this way need to be registered.

The registration file uses the following syntax for the application:

**** *AppName***.***ObjectName*[**.***VersionNumber*] **=** *human_readable_string*
**** *AppName***.***ObjectName***\CLSID = {***UUID***}**

AppName
The name of the application.

ObjectName
The name of the object to be registered (in this case, Application).

VersionNumber
The optional version number of the object.

human_readable_string
A string that describes the application to users. The recommended maximum length is 40 characters.

UUID
The universally unique ID for the application CLSID. To generate a UUID for your class, run the utility Guidgen.exe.

Registering Classes

Objects that can be created with **CoCreateInstance** must also be registered with the system. For these objects, registration maps a CLSID to the Automation component file (.dll or .exe) in which the object resides. The CLSID also maps an ActiveX object back to its application and ProgID.

The following figure shows how registration connects ProgIDs, CLSIDs, and ActiveX components.

{ewc msdncd, EWGraphic, bsd23524 1 /a "SDK_05.WMF"}

`\CLSID\TypeLib = {UUID of type library}`
The type library can be obtained from its CLSID.

`\CLSID\Programmable`
Indicates that the server is an ActiveX component.

These COM class registry keys are required. The following shows the resulting example code:

```
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1} = Hello 2.0
Application
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\ProgID =
Hello.Application.2
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\
VersionIndependentProgID = Hello.Application
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\LocalServer32
= hello.exe /Automation
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\TypeLib =
{F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\Programmable
```

The registration file uses the following syntax for each class of each object that the application exposes.

\CLSID\{*UUID***} =** *human_readable_string*
\CLSID\{*UUID***}*ProgID* **=** *AppName.ObjectName.VersionNumber*
\CLSID\{*UUID***}*VersionIndependentProgID* **=** *AppName.ObjectName*
\CLSID\{*UUID***}\LocalServer**[**32**] **=** *filepath*[*/***Automation**]
\CLSID\{*UUID***}\InProcServer**[**32**] **=** *filepath*[*/***Automation**]

human_readable_string
A string that describes the object to users. The recommended maximum length is 40 characters.

AppName
The name of the application, as specified previously in the application registration string.

ObjectName
The name of the object to be registered.

VersionNumber
The version number of the object.

UUID
The universally unique ID for the application CLSID. To generate a UUID for your class, run the utility Guidgen.exe.

filepath
The full path and name of the file that contains the object. The optional **/Automation** switch tells the application it was launched for Automation purposes. The switch should be specified for the Application object's class. For more information on **/Automation**, see "Initializing the Active Object" later in this chapter.

The ProgID and VersionIndependentProgID are used by other programmers to gain access to the objects

you expose. These IDs should use consistent naming guidelines across all your applications as follows:

- Can contain up to 39 characters.
- Must not contain any punctuation (except for the period)
- Must not start with a digit.

Version-independent names consist of an *AppName*.*ObjectName*, without a version number. For example, Word.Document or Excel.Chart.

Version-dependent names consist of an *AppName*.*ObjectName*.*VersionNumber*, such as Excel.Application.5.

LocalServer[**32**]
: Indicates that the ActiveX component is an .exe file and runs in a separate process from the ActiveX client. The optional 32 specifies a server intended for use on 32-bit Windows systems.

InProcServer[**32**]
: Indicates that the ActiveX component is a DLL and runs in the same process space as the ActiveX client. The optional 32 specifies a server intended for use on 32-bit Windows systems.

  The *filepath* you register should give the full path and name. Applications should not rely on the MS-DOS PATH variable to find the object.

Releasing OLE and Objects

To release OLE and the exposed objects, use the following functions:

- **RevokeActiveObject**–Ends an object's status as the active object.
- **CoRevokeClassObject**–Informs OLE that a class factory is no longer available for use by other applications.
- **OLEUninitialize**–Releases OLE.

Retrieving Objects

Automation provides several functions to identify and retrieve the active instance of an object or application, so you can make the object available to others.

- **RegisterActiveObject** – Sets the active object for an application. (Use when the application starts.)
- **RevokeActiveObject** – Revokes the active object. (Use when the application ends.)
- **GetActiveObject** – Retrieves a pointer to the active object. (In Visual Basic, this pointer is implemented by the **GetObject** function.)

Applications can have more than one active object at a time. To be initialized as active, an object must:

- Have a class factory (that is, the object provides an interface for creating instances of itself).
- Identify its class factory by a ProgID in the system registry.
- Be registered by a call to **RegisterActiveObject** when the object is created, or when it becomes active.

The Application object must be registered as an active object.

Returning Objects

To return an object from a property or method, the application should return a pointer to the object's implementation of the **IDispatch** interface. The data type of the return value should be VT_DISPATCH, or if the object does not support **IDispatch,** VT_UNKNOWN. The .odl file for the object should specify the name of the interface, rather than **IDispatch***, as follows:

```
ICustom * MyMember(...) {...};
```

The example declares a member that returns a pointer to a custom interface named **ICustom**.

Shutting Down Objects

ActiveX objects must shut down in the following way:

- If the object's application is visible, the object should shut down only in response to an explicit user command (for example, clicking **Exit** on the **File** menu) or the equivalent command from an ActiveX client.
- If the object's application is not visible, the object should shut down only when the last external reference is gone.
- If the object's application is visible and is controlled by an ActiveX client, it should become invisible when the user shuts it down (for example, clicking **Exit** on the **File** menu). This behavior allows the controller to continue to control the object. The controller should shut down only when the last external reference to the object has disappeared.

Application Design Considerations

When you expose objects to Automation, you need to decide which interfaces to implement and how to organize your objects. You should also create a type library. This section provides information to guide you in designing an Automation application.

Creating the Programmable Interface

An object's programmable interface comprises the properties, methods, and events that it defines. Organizing the objects, properties, and methods that an application exposes is like creating an object-oriented framework for an application. Chapter 4, "Standard Objects and Naming Guidelines," discusses some of the concepts behind naming and organizing the programmable elements that an application can expose.

Creating Methods

A *method* is an action that an object can perform, such as drawing a line or clearing the screen. Methods can take any number of arguments (including optional arguments), and they can be passed either by value or by reference. A method may or may not return a value.

Creating Properties

A *property* is a member function that sets or returns information about the state of the object, such as color or visibility. Most properties have a pair of accessor functions – a function to get the property value and a function to set the property value. Properties that are read-only or write-only, however, have only one accessor function.

Accessor Functions

The accessor functions for a single property have the same dispatch ID (DISPID). The purpose of each function is indicated by attributes that are set for the function. These attributes are set in the .odl file description of the function, and are passed in the *wFlags* parameter to **Invoke** in order to set the context for the call. The attributes and flags are shown in the following table.

Purpose of function	ODL attribute	wFlags
Returns a value.	**propget**	DISPATCH_PROPERTYGET
Sets a value.	**propput**	DISPATCH_PROPERTYPUT
Sets a reference.	**propputref**	DISPATCH_PROPERTYPUTREF

The **propget** attribute designates the accessor function that gets the value of a property. When the ActiveX client needs to get the value of the property, it passes the DISPATCH_PROPERTYGET flag to **Invoke**.

The **propput** attribute designates the accessor function that sets the value of a property. When an ActiveX client needs to set a property by value, it passes the DISPATCH_PROPERTYPUT flag to **Invoke**. In Visual Basic, **Let** statements set properties by value.

The **propputref** attribute indicates that the property should be set by reference, rather than by value. In these cases, ActiveX clients that need to set a reference to a property pass DISPATCH_PROPERTYPUTREF. Visual Basic treats the **Set** statement as a by-reference property assignment.

Implementing the Value Property

The **Value** property defines the default behavior of an object when no property or method is specified. It is typically used for the property that users associate most closely with the object. For example, a cell in a spreadsheet might have many properties (**Font, Width, Height,** and so on), but its **Value** property defines the value of the cell. To refer to this property, a user does not need to specify the property name `Cell(1,1).Value`**,** but can simply use `Cell(1,1)`.The **Value** property is identified by the dispatch ID named DISPID_VALUE. In an .odl file, the **Value** property for an object has the attribute **id(0)**.

Handling Events

In addition to supporting properties and methods, ActiveX objects can be a source of events. In Automation, an *event* is a method that is called by an ActiveX object, rather than implemented by the object. For example, an object might include an event method named Button that retrieves clicks of the mouse button. Instead of being implemented by the object, the Button method returns an object that is a source of events.

In Automation, you use the **source** attribute to identify a member that is a source of events.

Details of the Automation event interfaces are provided in *Programming with MFC*, provided with Visual C++ version 4.1 or later product documentation.

Creating the IUnknown Interface

The **IUnknown** interface defines three member functions that must be implemented for each object that is exposed. The prototypes for these functions reside in the header file, Ole2.h.

- **QueryInterface** – Identifies which OLE interfaces the object supports.
- **AddRef** – Increments a member variable that tracks the number of references to the object.
- **Release** – Decrements the member variable that tracks the instances of the object. If an object has zero references, **Release** frees the object.

These functions provide the fundamental interface through which OLE can access objects. The *Microsoft OLE Programmer's Guide and Reference* describes in detail how to implement the functions.

Creating the IDispatch Interface

The **IDispatch** interface provides a late-bound mechanism to access and retrieve information about an object's methods and properties. In addition to the member functions inherited from **IUnknown**, the following member functions should be implemented within the class definition of each object that will be exposed through Automation.

- **GetTypeInfoCount** – Returns the number of type descriptions for the object. For objects that support **IDispatch**, the type information count is always 1.
- **GetTypeInfo** – Retrieves a description of the object's programmable interface.
- **GetIDsOfNames** – Maps the name of a method or property to a dispatch ID, which can later be used to invoke the method or property.
- **Invoke** –Calls one of the object's methods, or gets or sets one of its properties.

You can implement **IDispatch** by any of the following means:

- Delegating to the **DispInvoke** and **DispGetIDsOfNames** functions, or to **ITypeInfo::Invoke** and **ITypeInfo::GetIDsOfNames**. This is the recommended approach, because it supports multiple locales and allows exceptions to be returned.
- Calling the **CreateStdDispatch** function. This approach is the simplest, but it does not provide for rich error handling or multiple national languages.
- Implementing the member functions without delegating to the dispatch functions. This approach is seldom necessary. Because **Invoke** is a complex interface with many subtle semantics that are difficult to emulate, it is strongly recommended that code delegate to **ITypeInfo::Invoke** to implement this mechanism.

Implementing Dual Interfaces

Although Automation allows you to implement an **IDispatch** interface, a VTBL interface, or a **dual** interface (which encompasses both), it is strongly recommended that you implement **dual** interfaces for all exposed ActiveX objects. **Dual** interfaces have significant advantages over **IDispatch**-only or VTBL-only interfaces.

- Binding can take place at compile time through the VTBL interface, or at run time through **IDispatch**.
- ActiveX clients that can use the VTBL interface may benefit from improved performance.
- Existing ActiveX clients that use the **IDispatch** interface will continue to work.
- The VTBL interface is easier to call from C++.
- **Dual** interfaces are required for compatibility with Visual Basic object support features.

Converting Existing Objects to Dual Interfaces

If you have already implemented exposed objects that support only the **IDispatch** interface, you should convert them to support **dual** interfaces. Use the following steps:

1. Edit the .odl file to declare a **dual** interface instead of an IDispatch-only interface.
2. Rearrange the parameter lists so that the methods and properties of your exposed objects return an HRESULT and pass their return values in a parameter.
3. If your object implements an exception handler, revise your code to use the Automation error handling interface. This interface provides detailed, contextual error information through both **IDispatch** and VTBL interfaces.

Registering Interfaces

Applications that add interfaces need to register the interfaces so OLE can find the appropriate remoting code for interprocess communication. By default, Automation registers dispinterfaces that appear in the .odl file. It also registers remote Automation-compatible interfaces that are not registered elsewhere in the system registry under the label **ProxyStubClsid32** (or **ProxyStubClsid** on 16-bit systems).

The information registered for an interface is as follows:

\Interface\{*IID***} =** *InterfaceName*
\Interface\{*IID***}\Typelib =** *LIBID*
\Interface\{*IID***}\ProxyStubClsid**[**32**] **=** *CLSID*

IID
The universally unique ID of the interface.

InterfaceName
The name of the interface.

LIBID
The universally unique ID associated with the type library in which the interface is described.

CLSID
The universally unique ID associated with the proxy/stub implementation of the interface, used internally by OLE for interprocess communication. ActiveX objects use the proxy/stub implementation of **IDispatch**.

Note To obtain a universally unique identifier (UUID), use the Guidgen.exe utility, which is a random number generator for creating unique identifiers. For more information about this utility, refer to Appendix E, "Managing GUIDs."

Creating Class Identifiers

Each object that is exposed for creation must have a unique class identifier (CLSID). Class IDs are universally unique identifiers (UUIDs, also called globally unique identifiers, or GUIDs) that identify class objects to OLE. The CLSID is included in an application, and must be registered with the operating system when an application is installed.

To generate UUIDs, run the Guidgen.exe utility. By default, Guidgen.exe puts a DEFINE_GUID macro on the Windows Clipboard, which can then be pasted into your source code.

Passing Formatted Data

Often, an application needs to accept formatted data as an argument to a method or property. Examples include a bitmap, formatted text, or a spreadsheet range. When handling formatted data, the application should pass an object that implements the OLE **IDataObject** interface. For detailed information about the interface, see the *Microsoft OLE Programmer's Guide and Reference*.

By using this interface, applications can retrieve the data of any Clipboard format. Because an **IDataObject** instance can provide data of more than one format, a caller can provide data in several formats, and let the called object choose which format is most appropriate.

If the data object implements **IDispatch**, it should be passed using the VT_DISPATCH flag. If the data object does not support **IDispatch**, it should be passed with the VT_UNKNOWN flag.

Implementing the IEnumVARIANT Interface

Automation defines the **IEnumVARIANT** interface to provide a standard way for ActiveX clients to iterate over collection objects. Every collection object must expose a read-only property named **_NewEnum** to let ActiveX clients know that the object supports iteration. The **_NewEnum** property returns an enumerator object that supports **IEnumVARIANT**.

The **IEnumVARIANT** interface provides a way to iterate through the items contained by a collection object. This interface is supported by an enumerator object that is returned by the **_NewEnum** property of the collection object, as in the following figure.

{ewc msdncd, EWGraphic, bsd23524 2 /a "SDK_02.WMF"}

The **IEnumVARIANT** interface defines these member functions:

- **Next** – Retrieves one or more elements in a collection, starting with the current element.
- **Skip** – Skips over one or more elements in a collection.
- **Reset** – Resets the current element to the first element in the collection.
- **Clone** – Copies the current state of the enumeration so you can return to the current element after using **Skip** or **Reset**.

Implementing the _NewEnum Property

The **_NewEnum** property identifies an object as supporting iteration through the **IEnumVARIANT** interface. This property has the following requirements:

- Must be named **_NewEnum** and must not be localized.
- Must return a pointer to the enumerator object's **IUnknown** interface.
- Must include DISPID = DISPID_NEWENUM (-4).

Type Libraries

You must create a type library for each set of exposed objects. Because VTBL references are bound at compile time, exposed objects that support VTBL binding must be described in a type library.

Type libraries provide these important benefits:

- Type checking can be performed at compile time. This may help developers of ActiveX clients to write fast, correct code to access objects.
- You can describe an interface with type information and implement **IDispatch::Invoke** for the interface using a single call to **DispInvoke**.
- Visual Basic applications can create objects with specific interface types, rather than the generic **Object** type, to take advantage of early binding.
- ActiveX clients that do not support VTBLs can read and cache DISPIDs at compile time, improving run-time performance.
- Type browsers can scan the library, allowing others to see the characteristics of objects.
- The **RegisterTypeLib** function can be used to register exposed objects in the registration database.
- The **UnRegisterTypeLib** function can be used to completely uninstall an application from the system registry.
- Local server access is improved because Automation uses information from the type library to package the parameters that are passed to an object in another process.

Creating a Type Library

Most ActiveX components create *type libraries*. Type libraries contain type information, Help file names and contexts, and function-specific documentation strings. Access to this information is available at both compile time and run time.

Type information is the Automation standard for describing the objects, properties, and methods exposed by the ActiveX component. Browsers and compilers use the type information to display and access the exposed objects.

Type libraries are described in Object Description Language (ODL) and are compiled by the MIDL compiler or the MkTypLib utility.

MIDL library statement example

```
[
    uuid(F37C8060-4AD5-101B-B826-00DD01103DE1),    // LIBID_Hello
    helpstring("Hello 2.0 Type Library"),
    lcid(0x0409),
    version(2.0)
]
library Hello
{
    importlib("stdole.tlb");
    [
        uuid(F37C8062-4AD5-101B-B826-00DD01103DE1),    // IID_IHello
        helpstring("Application object for the Hello application."),
        oleautomation,
        dual
    ]
    interface IHello : IDispatch
    {
        [propget, helpstring("Returns the application of the object.")]
        HRESULT Application([in, lcid] long localeID,
            [out, retval] IHello** retval)
    }
}
```

Object description script example

```
/* TDATA.ODL */
AppName library{
     dispinterface ObjeNamePro {
           interface ObjName
           }
}
```

Automation also supports the creation of alternative tools that compile and access type information. For information about creating these tools, refer to Chapter 9, "Type Description Interfaces."

A type library stores complete type information for all of an application's exposed objects. It may be included as a resource in a DLL or executable file, or remain as a stand-alone file (.tlb).

Quick Info

To create a type library

1. Write an object description script (.odl) file for the objects you expose.
2. Using the MIDL compiler or the MkTypLib utility, build the type library (.tlb) and class description

header (.h) file from the script.

Building a Type Library

The MIDL compiler and the MkTypLib utility build type libraries. These tools are described in Chapter 8, "Type Libraries and the Object Description Language."

Quick Info To create a type library from an object description script

Run the MIDL compiler or MkTypLib tool on the script. For example:

```
MIDL /TLB output.tlb /H output.h inscript.odl
```

or

```
MKTYPLIB /TLB output.tlb /H output.h inscript.odl
```

Based on the object description script `inscript.odl`, the example creates a type library named `output.tlb` and a header file named `output.h`.

After creating the type library, you can include it in the resource step of building your application, or leave it as a stand-alone file. In either case, be sure to specify the file name and path of the library in the application's registration (.reg) file, so Automation can find the type library when necessary. See the following section for information on registering the type library.

Quick Info To build an application that uses a type library

1. Include the header file in the project.
2. Compile the project.
3. Optionally, use the Resource Compiler (RC) to bind the type library with the compiled project. The type library can bind with DLLs or executable files. For example, to bind a type library named `output.tlb` with a DLL, use the following statement in the .rc file for the DLL:

   ```
   1 typelib output.tlb
   ```

   A DLL that contains a type library resource usually has the .olb (object library) extension.

The following figure illustrates the process.

{ewc msdncd, EWGraphic, bsd23524 3 /a "SDK_04.WMF"}

Registering a Type Library

Tools and applications that expose type information must register the information so that it is available to type browsers and programming tools. The correct registration entries for a type library can be generated by calling the **RegisterTypeLib** function on the type library. Regedit.exe, which is supplied with the Win32 SDK as well as Windows NT and Windows 95, can then be used to write the registration entries to a text file from the system registration database.

The following information is registered for a type library:

\TypeLib\{*libUUID***}**
\TypeLib\{*libUUID***}***major.minor* **=** *human_readable_string*
\TypeLib\{*libUUID***}***major.minor*\HELPDIR **=** [*helpfile_path*]
\TypeLib\{*libUUID***}***major.minor***Flags =** *typelib_flags*
\TypeLib\{*libUUID***}***major.minor**lcid**platform* **=** *localized_typelib_filename*

libUUID
The universally unique ID of the type library.

major.minor
The two-part version number of the type library. If only the minor version number increases, all the features of the previous type library are supported in a compatible way. If the major version number changes, code that compiled against the type library must be recompiled. The version number of the type library may differ from the version number of the application.

human_readable_string
A string that describes the type library to users. The recommended maximum length is 40 characters.

helpfile_path
The directory where the Help file for the types in the type library is located. If the application supports type libraries for multiple languages, the libraries may refer to different file names in the Help file directory.

typelib_flags
The hexadecimal representation of the type library flags for this type library. These are the values of the LIBFLAGS enumeration, and are the same flags specified in the *uLibFlags* parameter to **ICreateTypeLib::SetLibFlags**. These flags cannot have leading zeros or the *0x* prefix.

lcid
The hexadecimal string representation of the locale ID (LCID). It is one to four hexadecimal digits with no *0x* prefix and no leading zeros. The locale ID may have a neutral sublanguage ID.

platform
The target operating system platform: 16-bit Windows, 32-bit Windows, or Apple® Macintosh®.

localized_typelib_filename
The full name of the localized type library.

Using the locale ID specifier, an application can explicitly register the file names of type libraries for different languages. This allows the application to find the desired language without having to open all type libraries with a given name.

For example, to find the type library for Australian English (309), the application first looks for it. If that fails, the application looks for an entry for standard English (a primary identifier of 0x09). If there is no entry for standard English, the application looks for LANG_SYSTEM_DEFAULT (0). For more information on locale support, refer to your operating system documentation for the national language support (NLS) interface. For 16-bit systems, see Appendix A.

Example

```
; Type library registration information.
```

```
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0 =
Automation Hello 2.0 Type Library.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0\HELPDIR
=
; U.S. English.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0\9\win32
= hello.tlb
```

Returning an Error

ActiveX objects typically return rich contextual error information, which includes an error number, a description of the error, and the path of a Help file that supplies more information. Objects that do not need to return detailed error information can simply return an HRESULT that indicates the nature of the error.

Passing Exceptions Through IDispatch

When an error occurs, objects invoked through **IDispatch** can return DISP_E_EXCEPTION and pass the details in the *pexcepinfo* parameter (an EXCEPINFO structure) to **IDispatch::Invoke**. Refer to the EXCEPINFO structure in Chapter 5, "Dispatch Interface and API Functions," and see "Passing Exceptions Through VTBLs" later in this chapter.

"Hello" Sample

The Hello sample is an Automation application with one object. It has these characteristics:

- Supports VTBL binding.
- Permits multiple instances of its exposed object to exist at the same time.
- Implements **IErrorInfo** for exception handling.

This sample has been simplified for demonstration purposes. It has the following limitations:

- Has only one object.
- Uses only scalar argument types. Automation also supports methods and properties that accept arguments of complex types, including arrays, references to objects, and formatted data, but not structures.
- Supports one national language.

The sections that follow demonstrate how the Hello sample exposes a simple class. The code is abridged to illustrate the essential parts. For a complete listing, see the source code in the *Microsoft OLE Programmer's Guide and Reference* in the Win32 SDK.

Initializing OLE

When the Hello application starts, it initializes OLE and then creates the object to be exposed through Automation. For example (Main.cpp):

```
BOOL InitInstance (HINSTANCE hinst)
{
    HRESULT hr;
    TCHAR ach[STR_LEN];

    // Intialize OLE.
    hr = OleInitialize(NULL);
    if (FAILED(hr))
        return FALSE;

    // Create an instance of the Hello Application object. The object is
    // created with refcount 0.
    LoadString(hinst, IDS_HelloMessage, ach, sizeof(ach));
    hr = CHello::Create(hinst, ach, &g_phello);
    if (FAILED(hr))
        return FALSE;
    return TRUE;
}
```

This function calls **OleInitialize** to initialize OLE. It loads the string `ach` with the initial Hello message, obtained from the string table through the constant `IDS_HelloMessage`. Then it calls `CHello::Create` to create a single, global instance of the application object, passing it the initial Hello message and receiving a value for `g_phello`, a pointer to the instance. If the function is successful, it returns a value of **True**.

Registering the Active Object

After Hello creates an instance of the object, it exposes and registers the class factory (if necessary) and registers the active object (Main.cpp):

```
BOOL ProcessCmdLine(LPSTR pCmdLine,
```

```
                    LPDWORD pdwRegisterCF,
                    LPDWORD pdwRegisterActiveObject,
                    int nCmdShow)
{
    LPCLASSFACTORY pcf = NULL;
    HRESULT hr;

    *pdwRegisterCF = 0;
    *pdwRegisterActiveObject = 0;

    // Expose class factory for application object if command line
    // contains the /Automation switch.
    if (_fstrstr(pCmdLine, "-Automation") != NULL
        || _fstrstr(pCmdLine, "/Automation") != NULL)
    {
        pcf = new CHelloCF;
        if (!pcf)
            goto error;
        pcf->AddRef();
        hr = CoRegisterClassObject(CLSID_Hello, pcf,
                                   CLSCTX_LOCAL_SERVER,
                                   REGCLS_SINGLEUSE,
                                   pdwRegisterCF);
        if (hr != NOERROR)
            goto error;
        pcf->Release();
    }
    else g_phello->ShowWindow(nCmdShow); //Show if started stand-alone.

RegisterActiveObject(g_phello, CLSID_Hello, ACTIVEOBJECT_WEAK,
                    pdwRegisterActiveObject);
    return TRUE;

error:
    if (!pcf)
        pcf->Release();
    return FALSE;
}
```

The sample first checks the command line for the **/Automation** switch. This switch indicates that the application should be started for programmatic access, so that ActiveX clients can create additional instances of the application's class. In this case, the class factory must be created and registered. If the switch is present, the Hello sample creates a new CHelloCF object and calls its **AddRef** method, thereby creating the class factory.

Next, the sample calls **CoRegisterClassObject** to register the class factory. It passes the object's class ID (CLSID_Hello), a pointer to the CHelloCF object (pcf), and two constants (CLSCTX_LOCAL_SERVER and REGCLS_SINGLEUSE) that govern the class factory's use.

- CLSCTX_LOCAL_SERVER indicates that the executable code for the object runs in a separate process space from the controller.
- REGCLS_SINGLEUSE allows only one ActiveX client to use each instance of the class factory. The value returned through pdwRegisterCF must later be used to revoke the class factory.

The example specifies weak registration (ACTIVEOBJECT_WEAK), which means that OLE will release the

object when all external connections to it have disappeared. You should always give ActiveX objects weak registration. For more information, see "RegisterActiveObject" in Chapter 5, "Dispatch Interface and API Functions."

The *Microsoft OLE Programmer's Guide and Reference* provides more information on the functions **OleInitialize** and **CoRegisterClassObject**. *Inside OLE, Second Edition,* published by Microsoft Press, provides more information about verifying application entries in the registration database.

Registering the Hello Application

Finally, the sample registers the Hello application object in the running object table (ROT). Registering an active object allows ActiveX clients to retrieve an object that is already running, rather than create a new instance of the object. Use weak registration (ACTIVEOBJECT_WEAK) so that the running object table releases its reference when all external references are released. If strong registration is used (the default), the running object table will not release the reference until **RevokeActiveObject** is called. For more information, refer to Chapter 5, "Dispatch Interface and API Functions."

The following sample shows the registration entries for the Hello object.

```
REGEDIT
; Registration information for Automation Hello 2.0 Application.

; Version independent registration. Points to Version 2.0.
HKEY_CLASSES_ROOT\Hello.Application = Automation Hello Application
HKEY_CLASSES_ROOT\Hello.Application\Clsid = {F37C8061-4AD5-101B-B826-
00DD01103DE1}

; Version 2.0 registration.
HKEY_CLASSES_ROOT\Hello.Application.2 = Automation Hello 2.0 Application
HKEY_CLASSES_ROOT\Hello.Application.2\Clsid = {F37C8061-4AD5-101B-B826-
00DD01103DE1}
```

Implementing IDispatch

The **IDispatch** interface provides access to and information about an object. The interface requires the member functions **GetTypeInfoCount, GetTypeInfo, GetIdsOfNames,** and **Invoke.** The Hello sample implements **IDispatch** as follows (Hello.cpp):

```
STDMETHODIMP
CHello::GetTypeInfoCount(UINT FAR* pctinfo)
{
     *pctinfo = 1;
     return NOERROR;
}

STDMETHODIMP
CHello::GetTypeInfo(
           UINT itinfo,
           LCID lcid,
           ITypeInfo FAR* FAR* pptinfo)
{
     *pptinfo = NULL;

     if(itinfo != 0)
           return ResultFromScode(DISP_E_BADINDEX);

     m_ptinfo->AddRef();
```

```
      *pptinfo = m_ptinfo;

      return NOERROR;
}

STDMETHODIMP
CHello::GetIDsOfNames(
            REFIID riid,
            OLECHAR FAR* FAR* rgszNames,
            UINT cNames,
            LCID lcid,
            DISPID FAR* rgdispid)
{
      return DispGetIDsOfNames(m_ptinfo, rgszNames, cNames, rgdispid);
}

STDMETHODIMP
CHello::Invoke(
            DISPID dispidMember,
            REFIID riid,
            LCID lcid,
            WORD wFlags,
            DISPPARAMS FAR* pdispparams,
            VARIANT FAR* pvarResult,
            EXCEPINFO FAR* pexcepinfo,
            UINT FAR* puArgErr)
{
      {
            return DispInvoke(
            this, m_ptinfo,
            dispidMember, wFlags, pdispparams,
            pvarResult, pexcepinfo, puArgErr);
}
}
```

Automation includes two functions, **DispGetIdsOfNames** and **DispInvoke**, which provide standard implementations for **IDispatch::GetIDsOfNames**, and **IDispatch::Invoke**. The Hello sample uses these two functions to simplify the code.

Implementing IUnknown

Every OLE object must implement the **IUnknown** interface, which allows controllers to query the object to find out what interfaces it supports. **IUnknown** has three member functions: **QueryInterface**, **AddRef**, and **Release**. The Hello sample implements these functions for the CHello object as follows (Hello.cpp):

```
STDMETHODIMP
CHello::QueryInterface(REFIID iid, void FAR* FAR* ppv)
{
      *ppv = NULL;
      if (iid == IID_IUnknown || iid == IID_IDispatch || iid == IID_IHello
            *ppv = this;
      else if (iid == IID_ISupportErrorInfo)
            *ppv = &m_SupportErrorInfo;
      else return ResultFromScode(E_NOINTERFACE);

      AddRef();
```

```
     return NOERROR;
}

STDMETHODIMP_(ULONG)
CHello::AddRef(void)
{
     return ++m_cRef;
}

STDMETHODIMP_(ULONG)
CHello::Release(void)
{
if (--m_cRef == 0)
     {
           delete this;
           return 0;
     }
     return m_cRef;
}
```

Implementing IClassFactory

A class factory is a class that is capable of creating instances of another class. The Hello sample implements a single class factory named CHelloCF, as follows (HelloCf.cpp):

```
CHelloCF::CHelloCF(void)
{
     m_cRef = 0;
}

STDMETHODIMP
CHelloCF::QueryInterface(REFIID iid, void FAR* FAR* ppv)
{
     *ppv = NULL;
     if (iid == IID_IUnknown || iid == IID_IClassFactory)
           *ppv = this;
     else
           return ResultFromScode(E_NOINTERFACE);
     AddRef();
     return NOERROR;
}

STDMETHODIMP_(ULONG)
CHelloCF::AddRef(void)
{
     return ++m_cRef;
}

STDMETHODIMP_(ULONG)
CHelloCF::Release(void)
{
     if (--m_cRef == 0)
     {
           delete this;
           return 0;
```

```
    }
    return m_cRef;
}

STDMETHODIMP
CHelloCF::CreateInstance(IUnknown FAR* punkOuter,
                                    REFIID riid,
                                    void FAR* FAR* ppv)
{
    HRESULT hr;

    *ppv = NULL;

    // This implementation doesn't allow aggregation.
    if (punkOuter)
        return ResultFromScode(CLASS_E_NOAGGREGATION);

    hr = g_phello->QueryInterface(riid, ppv);
    if (FAILED(hr))
    {
        g_phello->Quit();
        return hr;
    }
    return NOERROR;
}
STDMETHODIMP
CHelloCF::LockServer(BOOL fLock)
{
    CoLockObjectExternal(g_phello, fLock, TRUE);
    return NOERROR;
}
```

The function **CHelloCF::CHelloCF** is a C++ constructor function. By default, the constructor function initializes the object's VTBLs; **CHelloCF::CHelloCF** also initializes the reference count for the class.

The class factory supports six member functions. **QueryInterface**, **AddRef**, and **Release** are the required **IUnknown** members, and **CreateInstance** and **LockServer** are the required **IClassFactory** members.

Implementing VTBL Binding

In addition to the **IDispatch** interface, the Hello sample supports VTBL binding. When a member is invoked, objects that support a VTBL interface return an HRESULT instead of a value, and pass their return value as the last parameter. Objects may also accept a locale ID parameter, which allows them to parse strings correctly for the local language. The following example shows how the **Visible** property is implemented (Hello.cpp):

```
STDMETHODIMP
CHello::put_Visible(BOOL bVisible)
{
    ShowWindow(bVisible ? SW_SHOW : SW_HIDE);
    return NOERROR;
}

STDMETHODIMP
CHello::get_Visible(BOOL FAR* pbool)
{
```

```
      *pbool =  m_bVisible;
      return NOERROR;
}
```

Additional information must be specified in the ODL to create a **dual** interface, as shown in "Creating Type Information" later in this chapter.

Registering the Interface for VTBL Binding

The following lines from the Hello.reg file register the interface for VTBL binding. In the example, **ProxyStubClsid** refers to the proxy and stub implementation of **IDispatch**.

```
HKEY_CLASSES_ROOT\Interface\{F37C8062-4AD5-101B-B826-00DD01103DE1} = IHello
HKEY_CLASSES_ROOT\Interface\{F37C806 2-4AD5-101B-B826-00DD01103DE1}\TypeLib
= {F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{F37C8062-4AD5-101B-B826-00DD01103DE1}\
ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}
```

Handling Errors

The Hello sample includes an exception handler that passes exceptions through **IDispatch::Invoke**, and supports rich error information through VTBLs (Hello.cpp):

```
STDMETHODIMP
CHello::RaiseException(int nID)
{
      extern return value g_scodes[];
      char szError[STR_LEN];
      ICreateErrorInfo *pcerrinfo;
      IErrorInfo *perrinfo;
      HRESULT hr;
      BSTR bstrDescription = NULL;

if (LoadString(g_phello->m_hinst,nID, szError, sizeof(szError)))
      bstrDescription = SysAllocString(TO_OLE_STRING(szError));

// Set ErrorInfo object so that VTBL binding controller can get
// rich error information. If the controller is using IDispatch
// to access properties or methods, DispInvoke will fill the
// EXCEPINFO structure using the values specified in the ErrorInfo
// object, and Dispinvoke will return DISP_E_EXCEPTION. The
// property or method must return a failure return value for DispInvoke
// to do this.
hr = CreateErrorInfo(hr))
{

      pcerrinfo->SetGUID(rguid);
      pcerrinfo->SetSource(g_phello->m_bstrProgID);
      if (bstrDescription)
            pcerrinfo->SetDescription(bstrDescription);
      hr = pcerrinfo->QueryInterface(IID_IErrorInfo,
      (LPVOID FAR*) &perrinfo);
      if (succeeded(hr))
      {

            SetErrorInfo(0,perrinfo);
```

```
            perrinfo->Release();
}

if (bstrDescription)
            SysFreeString(bstrDescription);
     return ResultFromScode(g_scodes[nID-1001]);
}
```

The member functions of the Hello sample call this routine when an exception occurs. **RaiseException** sets the system's error object so that controller applications that call through VTBLs can retrieve rich error information. Controllers that call through **IDispatch::Invoke** will be returned with this error information by **DispInvoke** through the EXCEPINFO structure.

Hello also implements the **ISupportErrorInfo** interface, which allows ActiveX clients to query whether an error object will be available (Hello.cpp):

```
CSupportErrorInfo::CSupportErrorInfo(IUnknown FAR* punkObject,
                                                    REFIID riid)
{
     m_punkObject = punkObject;
     m_iid = riid;
}

STDMETHODIMP
CSupportErrorInfo::QueryInterface(REFIID iid, void FAR* FAR* ppv)
{
     return m_punkObject->QueryInterface(iid, ppv);
}

STDMETHODIMP_(ULONG)
CSupportErrorInfo::AddRef(void)
{
     return m_punkObject->AddRef();
}

STDMETHODIMP_(ULONG)
CSupportErrorInfo::Release(void)
{
     return m_punkObject->Release();
}

STDMETHODIMP
CSupportErrorInfo::InterfaceSupportsErrorInfo(REFIID riid)
{
     return (riid == m_iid) ? NOERROR : ResultFromScode(S_FALSE);
}
```

Releasing Objects and OLE

When the Hello application ends, it revokes the class factory and the active object, and uninitializes OLE. For example (Main.cpp):

```
void Uninitialize(DWORD dwRegisterCF, DWORD dwRegisterActiveObject)
{
     if (dwRegisterCF != 0)
           CoRevokeClassObject(dwRegisterCF);
```

```
      if (dwRegisterActiveObject != 0)
            RevokeActiveObject(dwRegisterActiveObject, NULL);
      OleUninitialize();
}
```

Creating Type Information

Type information for the Hello sample is described in ODL. The MIDL compiler and MkTypLib use the .odl file to create a type library (Hellotl.tlb) and a header file (Hellotl.h).

The following example shows the description for the Hello type library, interface, and Application object (Hello.odl):

```
[
      uuid(F37C8060-4AD5-101B-B826-00DD01103DE1),      // LIBID_Hello.
      helpstring("Hello 2.0 Type Library"),
      lcid(0x009),
      version(2.0)
]
library Hello
{
      importlib("stdole32.tlb");
      [
      uuid(F37C8062-4AD5-101B-B826-00DD01103DE1),   // IID_Ihello.
      helpstring("Application object for the Hello application."),
      oleautomation,
      dual
      ]
      interface IHello : IDispatch
      {
            [propget, helpstring("Returns the application of the object.")]
            HRESULT Application([out, retval] IHello** retval);

      [propget,
            helpstring("Returns the full name of the application.")]
            HRESULT FullName([out, retval] BSTR* retval);

            [propget, id(0),
            helpstring("Returns the name of the application.")]
            HRESULT Name([out, retval] BSTR* retval);

            [propget, helpstring("Returns the parent of the object.")]
            HRESULT Parent([out, retval] IHello** retval);

            [propput]
            HRESULT Visible([in] boolean VisibleFlag);
            [propget,
            helpstring
            ("Sets or returns whether the main window is visible.")]
            HRESULT Visible([out, retval] boolean* retval);

            [helpstring("Exits the application.")]
            HRESULT Quit();

            [propput,
            helpstring("Sets or returns the hello message to be used.")]
```

```
            HRESULT HelloMessage([in] BSTR Message);
            [propget]
            HRESULT HelloMessage([out, retval] BSTR *retval);

            [helpstring("Say Hello using HelloMessage.")]
            HRESULT SayHello();
      }


      [
            uuid(F37C8061-4AD5-101B-B826-00DD01103DE1),      // CLSID_Hello.
            helpstring("Hello Class"),
            appobject
      ]
      coclass Hello
      {
            [default]         interface IHello;
                                    interface IDispatch;
      }
}
```

The items enclosed by square brackets are *attributes*, which provide further information about the objects in the file. The **oleautomation** and **dual** attributes, for example, indicate that the **IHello** interface supports both **IDispatch** and VTBL binding. The **appobject** attribute indicates that Hello is the Application object.

For more information about attributes, refer to Chapter 8, "Type Libraries and the Object Description Language."

Creating the Hello Registration File

The system registration database lists all the OLE objects in the system. OLE uses this database to locate objects and determine their capabilities. The registration file registers the application, the type library, and the exposed classes of the sample (Hello.reg):

```
REGEDIT
; Registration information for Automation Hello 2.0 Application.

; Version independent registration. Points to Version 2.0.
HKEY_CLASSES_ROOT\Hello.Application = Hello 2.0 Application
HKEY_CLASSES_ROOT\Hello.Application\Clsid = {F37C8061-4AD5-101B-B826-
00DD01103DE1}

; Version 2.0 registration
HKEY_CLASSES_ROOT\Hello.Application.2 = Hello 2.0 Application
HKEY_CLASSES_ROOT\Hello.Application.2\Clsid = {F37C8061-4AD5-101B-B826-
00DD01103DE1}
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1} = Hello 2.0
Application
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\ProgID =
Hello.Application.2
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\
VersionIndependentProgID = Hello.Application
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\LocalServer =
hello.exe /Automation
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\TypeLib =
{F37C8061-4AD5-101B-B826-00DD01103DE1}
```

```
HKEY_CLASSES_ROOT\CLSID\{F37C8061-4AD5-101B-B826-00DD01103DE1}\Programmable

; Type library registration information
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0 = Hello
2.0 Type Library
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0\HELPDIR
=
; English
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0\9\win32
= hello.tlb

; Interface registration. All interfaces that support vtable binding must be
; registered as follows. RegisterTypeLib & LoadTypeLib will do this
automatically.

; IID_IHello = {F37C8062-4AD5-101B-B826-00DD01103DE1}
; LIBID_Hello = {F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{F37C8062-4AD5-101B-B826-00DD01103DE1} = IHello
HKEY_CLASSES_ROOT\Interface\{F37C8062-4AD5-101B-B826-00DD01103DE1}\TypeLib =
{F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{F37C8062-4AD5-101B-B826-00DD01103DE1}\
ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}
```

To merge an object's registration information with the system registry, the object should expose the **DLLRegisterServer** API, as described in the *Microsoft OLE Programmer's Guide and Reference*. **DLLRegisterServer** should call **RegisterTypeLib** to register the type library and the interfaces supported by the application. This only applies to in-process servers. Out-of-process servers such as the Hello sample do not export **DLLRegisterServer**.

"Lines" Sample

The Lines sample is an ActiveX component application that implements collections. This sample file allows a collection of lines to be drawn on a pane using Automation. This sample implements the following features:

- **Dual** interfaces that allow access to automation properties and methods through VTBL binding and **IDispatch**.
- Rich error information for VTBL-binding controllers implemented by **ISupportErrorInfo** and **IErrorInfo**.
- Two collections.
- Active object registration using **RegisterActiveObject** and **RevokeActiveObject**.
- Correct shutdown behavior.
- A registration file that contains Lines.Application as the programmatic ID.
- Initial invisibility.

The following routine initializes OLE, and then creates an instance of the Lines Application object (Main.ccp):

```
BOOL InitInstance (HINSTANCE hinst)
{
     HRESULT hr;

     // Intialize OLE.
     hr = OleInitialize(NULL);
     if (FAILED(hr))
          return FALSE;

     // Create an instance of the Lines Application object. The object is
     // created with refcount 0.
     hr = CApplication::Create(hinst, &g_pApplication);
     if (FAILED(hr))
          return FALSE;
     return TRUE;
}
```

Initializing the Active Object

The following function creates and registers the application's class factory, and then registers the Lines Application object as the active object (Main.cpp):

```
BOOL ProcessCmdLine(LPSTR lpCmdLine, LPDWORD pdwRegisterCF,
                              LPDWORD pdwRegisterActiveObject, int nCmdShow)
{
     LPCLASSFACTORY pcf = NULL;
     HRESULT hr;
     *pdwRegisterCF = 0;
     *pdwRegisterActiveObject = 0;

     // Expose class factory for application object if command line
     // contains the /Automation switch.
     if (_fstrstr(lpCmdLine, "-Automation") != NULL
          || _fstrstr(lpCmdLine, "/Automation") != NULL)
     {
```

```
            pcf = new CApplicationCF;
            if (!pcf)
                  goto error;
            pcf->AddRef();
            hr = CoRegisterClassObject(CLSID_Lines, pcf,
                                                CLSCTX_LOCAL_SERVER,
REGCLS_SINGLEUSE,
                                                pdwRegisterCF);
            if (hr != NOERROR)
                  goto error;
            pcf->Release();
      }
      else                    // Show window if started as stand-alone.
      g_pApplication->ShowWindow(nCmdShow );

// Register Lines Application object in the running object table (ROT). //
Use weak registration so that the ROT releases its reference when
// all external references are released.
      RegisterActiveObject(g_pApplication, CLSID_Lines, ACTIVEOBJECT_WEAK,
                  pdwRegisterActiveObject);
      return TRUE;

error:
      if (pcf)
            pcf->Release();
      return FALSE;
}
```

Registering the Active Object

The sample application exposes the class factory for the Lines application, **CApplicationCF**, if the command line contains the **/Automation** switch. The switch indicates that the application was started for programmatic access, and therefore OLE needs to register the class factory and create an instance of the Application object. OLE applies this switch if it appears on the command line or in the application's registration file. OLE also supports the **/Embedding** switch, which indicates that an application has been started by a container application.

You should register the class factory for the Application object only if the application is launched with the **/Automation** switch. When **/Automation** is not specified, the application has been started for some reason other than programmatic access through Automation. If the class factory is registered under these circumstances, and a user later requests a new instance of the Application object, Automation will return the existing instance instead of creating a new one.

The sample calls **CoRegisterClassObject** to register the class factory as the active object. The CLSCTX_LOCAL_SERVER flag means the code that creates and manages Application objects will run in a separate process space.

Because the Application object's class factory is exposed, the call specifies the REGCLS_SINGLEUSE flag. When a multiple-document interface (MDI) application starts, it typically registers the class factory for its Document object, specifying REGCLS_MULTIPLEUSE. This flag, defined in the REGCLS enumeration, allows the existing application instance to be used later, when instances of the document objects need to be created. Each new Application object, however, requires a new instance of the application to be launched, and should therefore specify REGCLS_SINGLEUSE. If the application registered its class factory using REGCLS_MULTIPLEUSE, then the next **CreateObject** call that tries to create the application will get an existing copy.

In the following example, the macro defines a class ID for Lines (Tlb.h):

```
DEFINE_GUID(CLSID_Lines,0x3C591B21,0x1F13,0x101B,0xB8,0x26,0x00,0xDD,0x01,0x
10,0x3D,0xE1);
```

When the MIDL compiler ot MkTypLib creates the optional header file (Tlb.h), it inserts DEFINE_GUID macros for each library, interface, and each class in an application.

Creating the Lines Registration File

The registration file provides information about the application, the Application object, classes of objects, type libraries, and interfaces. Entries for objects and interfaces start with the constant HKEY_CLASSES_ROOT, which represents the root key of the entire registration database. Entries for type libraries start with HKEY_TYPELIB_ROOT. After the constant, each entry supplies specific information about an object, type library, or interface.

Use the following steps to create the registration file:

1. Copy the file Lines.reg.
2. Rename and edit this file, adding entries for the application.

Registering the Lines Application Files

The Lines sample uses the following entries to register its Application object (Lines.Application) and its type library with the system (Lines.reg):

```
REGEDIT
; Registration information for the Lines Application object. Version
independent registration.

HKEY_CLASSES_ROOT\Lines.Application = Lines
HKEY_CLASSES_ROOT\Lines.Application\Clsid = {3C591B21-1F13-101B-B826-
00DD01103DE1}

; Version 1.0 registration
HKEY_CLASSES_ROOT\Lines.Application.1 = Lines 1.0
HKEY_CLASSES_ROOT\Lines.Application.1\Clsid = {3C591B21-1F13-101B-B826-
00DD01103DE1}

HKEY_CLASSES_ROOT\CLSID\{3C591B21-1F13-101B-B826-00DD01103DE1} = Lines 1.0
HKEY_CLASSES_ROOT\CLSID\{3C591B21-1F13-101B-B826-00DD01103DE1}\ProgID =
Lines.Application.1
HKEY_CLASSES_ROOT\CLSID\{3C591B21-1F13-101B-B826-00DD01103DE1}\
VersionIndependentProgID = Lines.Application
HKEY_CLASSES_ROOT\CLSID\{3C591B21-1F13-101B-B826-00DD01103DE1}\LocalServer32
= lines.exe /Automation
HKEY_CLASSES_ROOT\CLSID\{3C591B21-1F13-101B-B826-00DD01103DE1}\TypeLib =
{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\CLSID\{3C591B21-1F13-101B-B826-00DD01103DE1}\Programmable

; Type library registration information.
HKEY_CLASSES_ROOT\TypeLib\{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\TypeLib\{3C591B20-1F13-101B-B826-00DD01103DE1}\1.0 = Lines
1.0 Type Library
HKEY_CLASSES_ROOT\TypeLib\{3C591B20-1F13-101B-B826-00DD01103DE1}\1.0\HELPDIR
=
;English
```

```
HKEY_CLASSES_ROOT\TypeLib\{3C591B20-1F13-101B-B826-00DD01103DE1}\1.0\9\win32
= lines.tlb

; Interface registration. All interfaces that support VTBL binding must be
; registered as follows. RegisterTypeLib will do this automatically.

; LIBID_Lines = {3C591B20-1F13-101B-B826-00DD01103DE1}

; IID_IPoint = {3C591B25-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B25-1F13-101B-B826-00DD01103DE1} = IPoint
HKEY_CLASSES_ROOT\Interface\{3C591B25-1F13-101B-B826-00DD01103DE1}\TypeLib =
{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B25-1F13-101B-B826-00DD01103DE1}\
ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}

; IID_ILine = {3C591B24-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B24-1F13-101B-B826-00DD01103DE1} = ILine
HKEY_CLASSES_ROOT\Interface\{3C591B24-1F13-101B-B826-00DD01103DE1}\TypeLib =
{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B24-1F13-101B-B826-00DD01103DE1}\
ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}

; IID_ILines = {3C591B26-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B26-1F13-101B-B826-00DD01103DE1} = ILines
HKEY_CLASSES_ROOT\Interface\{3C591B26-1F13-101B-B826-00DD01103DE1}\TypeLib =
{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B26-1F13-101B-B826-00DD01103DE1}\
ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}

; IID_IPoints = {3C591B27-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B27-1F13-101B-B826-00DD01103DE1} = IPoints
HKEY_CLASSES_ROOT\Interface\{3C591B27-1F13-101B-B826-00DD01103DE1}\TypeLib =
{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B27-1F13-101B-B826-00DD01103DE1}\
ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}

; IID_IPane = {3C591B23-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B23-1F13-101B-B826-00DD01103DE1} = IPane
HKEY_CLASSES_ROOT\Interface\{3C591B23-1F13-101B-B826-00DD01103DE1}\TypeLib =
{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B23-1F13-101B-B826-00DD01103DE1}\
ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}

; IID_IApplication = {3C591B22-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B22-1F13-101B-B826-00DD01103DE1} =
IApplication
HKEY_CLASSES_ROOT\Interface\{3C591B22-1F13-101B-B826-00DD01103DE1}\TypeLib =
{3C591B20-1F13-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\Interface\{3C591B22-1F13-101B-B826-00DD01103DE1}\ProxySt
```

Creating the IUnknown Interface for the Lines Application

The **IUnknown** interface for the Lines object looks like this (Lines.cpp):

```
STDMETHODIMP
```

```
CLine::QueryInterface(REFIID iid, void FAR* FAR* ppv)
{
      *ppv = NULL;

      if (iid == IID_IUnknown || iid == IID_IDispatch || iid == IID_ILine)
            *ppv = this;
      else if (iid == IID_ISupportErrorInfo)
            *ppv = &m_SupportErrorInfo;
      else return ResultFromScode(E_NOINTERFACE);

      AddRef();
      return NOERROR;
}

STDMETHODIMP_(ULONG)
CLine::AddRef(void)
{
      return ++m_cRef;
}

STDMETHODIMP_(ULONG)
CLine::Release(void)
{
      if(--m_cRef == 0)
      {
            delete this;
            return 0;
      }
      return m_cRef;
}
```

Creating the IDispatch Interface for the Lines Application

The following sections explain how to to implement **IDispatch** by using **CreateStdDispatch** and **DispInvoke**.

Implementing IDispatch by Calling CreateStdDispatch

The simplest way to implement the **IDispatch** interface is to call **CreateStdDispatch**. This approach works for ActiveX objects that return only the standard dispatch exception codes, support a single national language, and do not support dual interfaces.

CreateStdDispatch returns a pointer to the created **IDispatch** interface. It takes three pointers as input: a pointer to the object's **IUnknown** interface, a pointer to the object to expose, and a pointer to the type information for the object. The following example implements **IDispatch** for an object named CCalc by calling **CreateStdDispatch** on the loaded type information:

```
CCalc FAR*
CCalc::Create()
{
      HRESULT hresult;
      CCalc FAR* pcalc;
      ITypeLib FAR* ptlib;
      ITypeInfo FAR* ptinfo;
      IUnknown FAR* punkStdDisp;
```

```
      ptlib = NULL;
      ptinfo = NULL;

      // Some error handling code omitted.
      if ((pcalc = new FAR CCalc()) == NULL)
            return NULL;
      pcalc->AddRef();

      // Load the type library from the information in the registry.
      if ((hresult = LoadRegTypeLib(LIBID_DspCalc2, 1, 0, 0x0409, &ptlib))
            !=    NOERROR){
            goto LError0;
      }
      if ((hresult = ptlib->GetTypeInfoOfGuid(IID_ICalculator, &ptinfo))
            != NOERROR){
            goto LError0;
      }

      // Create an aggregate with an instance of the default
      // implementation of IDispatch that is initialized with our
      // TypeInfo.
      //
      hresult = CreateStdDispatch(
                  pcalc,                                    // Controlling
unknown.
                  &(pcalc->m_arith),                  // VTBL pointer to
dispatch on.
                  ptinfo,
                  &punkStdDisp);
```

Implementing IDispatch by Delegating

Another way to implement **IDispatch** is to use the dispatch functions **DispInvoke** and **DispGetIDsOfNames**. These functions give you the option of supporting multiple national languages and creating application-specific exceptions that are passed back to ActiveX clients.

The Lines sample implements **IDispatch::GetIDsOfNames** and **IDispatch::Invoke** using these functions (Lines.cpp):

```
STDMETHODIMP
CLines::GetIDsOfNames(
            REFIID riid,
            char FAR* FAR* rgszNames,
            UINT cNames,
            LCID lcid,
            DISPID FAR* rgdispid)
{
      return DispGetIDsOfNames(m_ptinfo, rgszNames, cNames, rgdispid);
}

STDMETHODIMP
CLines::Invoke(
            DISPID dispidMember,
            REFIID riid,
            LCID lcid,
            WORD wFlags,
```

```
            DISPPARAMS FAR* pdispparams,
            VARIANT FAR* pvarResult,
            EXCEPINFO FAR* pexcepinfo,
            UINT FAR* puArgErr)
{

      {
      return DispInvoke(
            this, m_ptinfo,
            dispidMember, wFlags, pdispparams,
            pvarResult, pexcepinfo, puArgErr);
}
}
```

The Lines object implements the **IID_ILine** **dual** interface for VTBL binding. It also implements the **IID_ISupportErrorInfo** interface so that it can return rich, contextual error information through VTBLs.

Implementing the Class Factory for the Lines Application

The Lines sample implements a class factory for its Application object, as follows (Appcf.cpp):

```
STDMETHODIMP
CApplicationCF::CreateInstance(IUnknown FAR* punkOuter,
                                                REFIID riid,
                                                void FAR* FAR* ppv)
{
      HRESULT hr;

      *ppv = NULL;

      // This implementation doesn't allow aggregation.
      if (punkOuter)
            return ResultFromScode(CLASS_E_NOAGGREGATION);

      // This is REGCLS_SINGLEUSE class factory, so CreateInstance will be
      // called at most once. An application object has a REGCLS_SINGLEUSE
      // class factory. The global application object has already been
      // created when CreateInstance is called. A REGCLS_MULTIPLEUSE class
      // factory's CreateInstance would be called multiple times and would
      // create a new object each time. An MDI application would have a
      // REGCLS_MULTIPLEUSE class factory for its document objects.

      hr = g_pApplication->QueryInterface(riid, ppv);
      if (FAILED(hr))
      {
            g_pApplication->Quit();
            return hr;
      }
      return NOERROR;
}

STDMETHODIMP
CApplicationCF::LockServer(BOOL fLock)
{
      CoLockObjectExternal(g_pApplication, fLock, TRUE);
      return NOERROR;
```

```
}
```

The object's class factory must also implement an **IUnknown** interface. For example (Appcf.cpp):

```
STDMETHODIMP
CApplicationCF::QueryInterface(REFIID iid, void FAR* FAR* ppv)
{
      *ppv = NULL;

      if (iid == IID_IUnknown || iid == IID_IClassFactory)
            *ppv = this;
      else
            return ResultFromScode(E_NOINTERFACE);
      AddRef();
      return NOERROR;
}

STDMETHODIMP_(ULONG)
CApplicationCF::AddRef(void)
{
      return ++m_cRef;
}

STDMETHODIMP_(ULONG)
CApplicationCF::Release(void)
{
      if(--m_cRef == 0)
      {
            delete this;
            return 0;
      }
      return m_cRef;
}
```

Setting Up the VTBL Interface

The Lines sample supports VTBL binding as well as the **IDispatch** interface. By supporting this **dual** interface, the sample allows ActiveX clients both the flexibility of the **IDispatch** interface and the speed of VTBLs. Controllers that know the names of the members can compile directly against the function pointers in the VTBL. Controllers that do not have this information can use **IDispatch** at run time.

To have a **dual** interface, an interface must:

- Declare all of its members to return an HRESULT, and pass their actual return values as the last parameter.
- Have only Automation-compatible parameters and return types, as described in Chapter 7, "Conversion and Manipulation Functions."
- Specify the **dual** attribute on the interface description in the .odl file.
- Initialize the VTBLs with the appropriate member function pointers.

In the Lines sample, the interfaces **IPoint, IPoints, ILine, ILines, IPane,** and **IApplication** are all dual interfaces. The **IPoint** interface defines functions that get and put the values of the **X** and **Y** properties, as follows (Point.cpp):

```
STDMETHODIMP
```

```
CPoint::get_x(int FAR* pnX)
{
     *pnX = m_nX;
     return NOERROR;
}

STDMETHODIMP
CPoint::put_x(int nX)
{
     m_nX = nX;
     return NOERROR;
}

STDMETHODIMP
CPoint::get_y(int FAR* pnY)
{
     *pnY = m_nY;
     return NOERROR;
}

STDMETHODIMP
CPoint::put_y(int nY)
{
     m_nY = nY;
     return NOERROR;
}
```

The **get_x** and **get_y** accessor functions pass their return values in the last parameter, *pnX* and *pnY*, and return an HRESULT as the function value.

In the .odl file, the interface is described as follows (Lines.odl):

```
[
            uuid(3C591B25-1F13-101B-B826-00DD01103DE1),    // IID_IPoint.
            helpstring("Point object."),
            oleautomation,
            dual
      ]
      interface IPoint : IDispatch
      {
            [propget, helpstring("Returns and sets x coordinate.")]
            HRESULT x([out, retval] int* retval);
            [propput, helpstring("Returns and sets x coordinate.")]
            HRESULT x([in] int Value);

            [propget, helpstring("Returns and sets y coordinate.")]
            HRESULT y([out, retval] int* retval);
            [propput, helpstring("Returns and sets y coordinate.")]
            HRESULT y([in] int Value);
      }
```

The attributes **oleautomation** and **dual** indicate that the interface supports both **IDispatch** and VTBL binding. All of the member functions are declared with HRESULT return values. The **Get** accessor functions, indicated by the **propget** attribute, return their value in the last parameter. This parameter has the **out** and **retval** attributes.

In the Lines sample, the Application object exposes the following method (App.cpp):

```
STDMETHODIMP
CApplication::CreatePoint(IPoint FAR* FAR* ppPoint)
{
     CPoint FAR* ppoint = NULL;
     HRESULT hr;

     // Create new item and QI for IDispatch.
     hr = CPoint::Create(&ppoint);
     if (FAILED(hr))
          {hr = RaiseException(IDS_OutOfMemory); goto error;}

     hr = ppoint->QueryInterface(IID_IDispatch, (void FAR* FAR*)ppPoint);
     if (FAILED(hr))
          {hr = RaiseException(IDS_Unexpected); goto error;}
     return NOERROR;

error:
     if (ppoint)
          delete ppoint;
     return hr;
}
```

The **CreatePoint** method creates a new point and returns a pointer to it in the parameter pPoint.

In the Lines sample, the CLine object exposes the **Color** property. This property is implemented by the following accessor functions (Lines.cpp):

```
STDMETHODIMP
CLine::get_Color(long FAR* plColorref)
{
     *plColorref =  m_colorref;
     return NOERROR;
}
STDMETHODIMP
CLine::put_Color(long lColorref)
{
     m_colorref = (COLORREF)lColorref;
     return NOERROR;
}
```

Implementing the Value Property

In the Lines sample, ILines.Item, IPoints.Item, and IApplication.Name are the Value properties of the objects ILines, IPoints, and IApplication, respectively. The ILines.Item object is described as follows:

```
interface ILines : IDispatch
     {
.
.// Some descriptions omitted for brevity.
.
     [propget, id(0), helpstring(
"Given an integer index, returns one of the lines in the collection")]
     HRESULT Item([in] long Index,[out, retval] ILine** retval);
.
```

```
}
```

Using this property, a user can refer to the fourth line in the collection as **ILines(4).Item** or simply as **ILines(4)**.

For more information on recommended objects, properties, and methods, see Chapter 4, "Standard Objects and Naming Guidelines."

Restricting Access to Objects

Automation provides several ways of restricting access to objects. The simplest approach is not to document the properties and methods you do not want users to see. Alternatively, you can prevent a property or method from appearing in type library browsers by specifying the **hidden** attribute in the .odl file.

The **restricted** attribute goes one step further, preventing user calls from binding to the property or method, as well as hiding it from type browsers. For example, the following restricts access to the **_NewEnum** property of the ILines object:

```
[propget, restricted, id(DISPID_NEWENUM)]            // Must be propget.
     HRESULT _NewEnum( [out, retval] IUnknown** retval);
```

Restricted properties and methods can be invoked by ActiveX clients, but are not visible to the user who may be using a language such as Visual Basic. In addition, they cannot be bound to by user calls.

Creating Collection Objects

A collection object contains a group of exposed objects of the same type and can iterate over them. Collection objects do not need an **IClassFactory** implementation, because they are accessed from elements that have their own class factories.

For example, the Lines sample has a collection object named CLines that iterates over a group of Line objects. The following routine creates and initializes the CLines collection object (Lines.cpp):

```
STDMETHODIMP
CLines::Create(ULONG lMaxSize, long lLBound, CPane FAR* pPane,
                        CLines FAR* FAR* ppLines)
{
     HRESULT hr;
     CLines FAR* pLines = NULL;
     SAFEARRAYBOUND sabound[1];

     *ppLines = NULL;

     // Create new collection.
     pLines = new CLines();
     if (pLines == NULL)
           goto error;

     pLines->m_cMax = lMaxSize;
     pLines->m_cElements = 0;
     pLines->m_lLBound = lLBound;
     pLines->m_pPane = pPane;

     // Load type information for the Lines collection from type library.
     hr = LoadTypeInfo(&pLines->m_ptinfo, IID_ILines);
     if (FAILED(hr))
```

```
            goto error;

      // Create a safe array of variants used to implement the collection.
      sabound[0].cElements = lMaxSize;
      sabound[0].lLbound = lLBound;
      pLines->m_psa = SafeArrayCreate(VT_VARIANT, 1, sabound);
      if (pLines->m_psa == NULL)
      {
            hr = ResultFromScode(E_OUTOFMEMORY);
            goto error;
      }

      *ppLines = pLines;
      return NOERROR;

error:
      if (pLines == NULL)
            return ResultFromScode(E_OUTOFMEMORY);
      if (pLines->m_ptinfo)
            pLines->m_ptinfo->Release();
      if (pLines->m_psa)
            SafeArrayDestroy(pLines->m_psa);

      pLines->m_psa = NULL;
      pLines->m_ptinfo = NULL;

      delete pLines;
      return hr;
}
```

The parameters to **CLines::Create** specify the maximum number of lines that the collection can contain, the lower bound of the indexes of the collection, and a pointer to a pane, which contains the lines and points in the sample.

Implementing the IEnumVARIANT Interface for the Lines Application

In the Lines sample, **CEnumVariant** implements the **Next, Skip, Reset,** and **Clone** member functions (Enumvar.cpp):

```
STDMETHODIMP
CEnumVariant::Next(ULONG cElements, VARIANT FAR* pvar,
                           ULONG FAR* pcElementFetched)
{
      HRESULT hr;
      ULONG l;
      long l1;
      ULONG l2;

      if (pcElementFetched != NULL)
            *pcElementFetched = 0;

      // Retrieve the elements of the next cElements.
      for (l1=m_lCurrent, l2=0; l1<(long)(m_lLBound+m_cElements) &&
            l2<cElements; l1++, l2++)
      {
            hr = SafeArrayGetElement(m_psa, &l1, &pvar[l2]);
```

```
            if (FAILED(hr))
                  goto error;
      }
      // Set count of elements retrieved.
      if (pcElementFetched != NULL)
            *pcElementFetched = l2;
      m_lCurrent = l1;

      return (l2 < cElements) ? ResultFromScode(S_FALSE) : NOERROR;
error:
      for (l=0; l<cElements; l++)
            VariantClear(&pvar[l]);
      return hr;
}

STDMETHODIMP
CEnumVariant::Skip(ULONG cElements)
{
      m_lCurrent += cElements;
      if (m_lCurrent > (long)(m_lLBound+m_cElements))
      {
            m_lCurrent =  m_lLBound+m_cElements;
            return ResultFromScode(S_FALSE);
      }
      else return NOERROR;
}

STDMETHODIMP
CEnumVariant::Reset()
{
      m_lCurrent = m_lLBound;
      return NOERROR;
}

STDMETHODIMP
CEnumVariant::Clone(IEnumVARIANT FAR* FAR* ppenum)
{
      CEnumVariant FAR* penum = NULL;
      HRESULT hr;

      *ppenum = NULL;

      hr = CEnumVariant::Create(m_psa, m_cElements, &penum);
      if (FAILED(hr))
            goto error;
      penum->AddRef();
      penum->m_lCurrent = m_lCurrent;

      *ppenum = penum;
      return NOERROR;
error:
      if (penum)
            penum->Release();
      return hr;
}
```

Implementing the _NewEnum Property for the Lines Application

The Lines sample contains two collections, Lines and Points, and implements a **_NewEnum** property for each. Both are restricted properties, available to ActiveX clients, but invisible to users of scripting or macro languages supported by ActiveX clients. The property returns an enumerator (**IEnumVARIANT**) for the items in the collection.

The following code implements the **_NewEnum** property for the Lines collection (Lines.cpp):

```
STDMETHODIMP
CLines::get__NewEnum(IUnknown FAR* FAR* ppunkEnum)
{
     CEnumVariant FAR* penum = NULL;;
     HRESULT hr;

     *ppunkEnum = NULL;

     // Create a new enumerator for items currently in the collection and
     // QueryInterface for IUnknown.
     hr = CEnumVariant::Create(m_psa, m_cElements, &penum);
     if (FAILED(hr))
          {hr = RaiseException(IDS_OutOfMemory); goto error;}
     hr = penum->QueryInterface(IID_IUnknown, (VOID FAR* FAR*)ppunkEnum);
     if (FAILED(hr))
          {hr = RaiseException(IDS_Unexpected); goto error;}
     return NOERROR;

error:
     if (penum)
          delete penum;
     return hr;
}
```

Returning Errors

The Lines sample defines an exception handler that fills the EXCEPINFO structure and signals **IDispatch** to return DISP_E_EXCEPTION (App.cpp):

```
STDMETHODIMP
HRESULT RaiseException (int nID, Refguid rguid)
{
     extern return value g_scodes[];
     TCHAR szError[STR_LEN];
     ICreateErrorInfo *pcerrinfo;
     IErrorInfo *perrinfo;
     HRESULT hr;
     BSTR bstrDescription = NULL;

     if (LoadString(g_pApplication->m_hinst, nID, szError,
     sizeof(szError)));
     bstrDescription = SysAllocString(TO_OLE_STRING(szError));
```

```
      // Set ErrInfo object so that VTBL binding controllers can get
      // rich error information. If the controller is using IDispatch to
      // access properties or methods, DispInvoke will fill the EXCEPINFO
      // structure using the values specified in the ErrorInfo object and
      // DispInvoke will return DISP_e_EXCEPTION. The property or method
      // must return a failure return value for DispInvoke to do this.
            hr = CreateErrorInfo(&pcerrinfo);
            if (SUCCEEDED(hr))
      {
            pcerrinfo->SetGUID(rguid);
            pcerrinfo->SetSource(g_pApplication->m_bstrProgID);
            if (bstrDescription)
                  pcerrinfo->SetDescription(bstrDescription);
            hr = pcerrinfo->QueryInterface(IID_IerrorInfo, (LPVOID FAR*)
                        &perrinfo);
            if (SUCCEEDED(hr))
{
                  SetErrorInfo(0, perrinfo);
                  perrinfo->Release();
      }
      if (bstrDescription)
            SysFreeString(bstrDescription);
      return ResultFromScode(g_scodes[nID-1001]);
}
```

Properties and methods in the Lines sample call this routine when an exception occurs. **RaiseException** sets the system's error object, so that controller applications that call through VTBLs can retrieve rich error information. Controllers that call through **IDispatch::Invoke** will be returned with this error information by **DispInvoke** through the EXCEPINFO structure.

Passing Exceptions Through VTBLs

The Lines sample also provides rich error information for members invoked through VTBLs. Because VTBL-bound calls bypass the **IDispatch** interface, they cannot return exceptions through **IDispatch**. Instead, they must use the error handling interfaces in Automation. The **RaiseException** function shown in the example calls **CreateErrorInfo** to create an error object, then fills the object's data fields with information about the error. When all of the information has been successfully recorded, it calls **SetErrorInfo** to associate the error object with the current thread of execution.

ActiveX objects similar to the collection object (CApplication), which use the error interfaces, must also implement the **ISupportErrorInfo** interface. This interface identifies the object as supporting the error interfaces, and ensures that error information can be propagated correctly up the call chain. The following example shows how the Lines sample implements this interface (Errinfo.cpp):

```
STDMETHODIMP
CSupportErrorInfo::CSupportErrorInfo(IUnknown FAR* punkObject,
            REFIID riid)
{
      m_punkObject = punkObject;
      m_iid = riid;
}

CSupportErrorInfo::QueryInterface(REFIID iid, void FAR* FAR* ppv)
{
      return m_punkObject->QueryInterface(iid, ppv);
}
```

```
STDMETHODIMP_(ULONG)
CSupportErrorInfo::AddRef(void)
{
     return m_punkObject->AddRef();
}

STDMETHODIMP_(ULONG)
CSupportErrorInfo::Release(void)
{
     return m_punkObject->Release();
}

STDMETHODIMP
CSupportErrorInfo::InterfaceSupportsErrorInfo(REFIID riid)
{
     return (riid == m_iid) ? NOERROR : ResultFromScode(S_FALSE);
}
```

ISupportErrorInfo has the **QueryInterface**, **AddRef**, and Release methods inherited from the **IUnknown** interface, along with the **InterfaceSupportsErrorInfo** method. ActiveX clients call **InterfaceSupportsErrorInfo** to check whether the ActiveX object supports the **IErrorInfo** interface, so they can access the error object. For details, see Chapter 11, "Error Handling Interfaces."

Releasing OLE on Exit

The following code revokes an active Lines object, revokes the Lines class, and then uninitializes OLE (Main.cpp):

```
void Uninitialize(DWORD dwRegisterCF, DWORD dwRegisterActiveObject)
{
     if (dwRegisterCF != 0)
          CoRevokeClassObject(dwRegisterCF);
     if (dwRegisterActiveObject != 0)
          RevokeActiveObject(dwRegisterActiveObject, NULL);
     OleUninitialize();
}
```

Writing an Object Description Script

An object description script is essentially an annotated header file, written in Object Description Language. The following example shows a portion of Lines.odl, the object description script for the Lines sample.

```
[
     uuid(3C591B20-1F13-101B-B826-00DD01103DE1),          // LIBID_Lines.
     helpstring("Lines 1.0 Type Library"),
     lcid(0x09),
     version(1.0)
]

library Lines
{
     importlib("stdole.tlb");
     #define DISPID_NEWENUM -4
.
```

.
.

The preceding entry describes the type library (Lines.tlb) created by the sample. The items in square brackets are attributes, which provide additional information about the library. In the example, the attributes give the library's universally unique identifier (UUID), a Help string, a locale identifier, and a version number.

The **importlib** directive is similar to the C or C++ **#include** directive. It allows access to the type descriptions in the file Stdole.tlb from the Lines library. However, it does not copy those types into the Lines.tlb. To use Lines.tlb, both the Lines.tlb and Stdole.tlb files must be available.

By default, .odl files are preprocessed with the C preprocessor, so the **#include** and **#define** directives can be used.

The .odl file script continues with information on the objects in the type library:

```
      [
            uuid(3C591B25-1F13-101B-B826-00DD01103DE1),           //
 IID_Ipoint.
            helpstring("Point object."),
            oleautomation,
            dual
      ]
      interface IPoint : IDispatch
      {
            [propget, helpstring("Returns and sets x coordinate.")]
            HRESULT x( [out, retval] int* retval);
            [propput, helpstring("Returns and sets x coordinate.")]
            HRESULT x([in] int Value);

            [propget, helpstring("Returns and sets y coordinate.")]
            HRESULT y( [out, retval] int* retval);
            [propput, helpstring("Returns and sets y coordinate.")]
            HRESULT y([in] int Value);
      }
      // .
      // Additional definitions omitted for brevity.
      // .
}
```

This entry describes the **IPoint** interface. The interface has the attributes **oleautomation** and **dual**, indicating that the types of all its properties and methods are compatible with Automation, and that it supports binding through both **IDispatch** and VTBLs. The **IPoint** interface has two pairs of property accessor functions, which set and return the **X** and **Y** properties.

The **Value** parameter of both the **X** and **Y** properties has the **in** attribute. These parameters supply a value and are read-only. Conversely, the *retval* parameter of each property has the **out** and **retval** attributes, indicating that it returns the value of the property.

Because **IPoint** supports VTBL binding and rich error information, its properties return HRESULTs and pass their function return values through *retval* parameters. For more information, see Chapter 8, "Type Libraries and the Object Description Language."

```
[
```

```
            uuid(3C591B21-1F13-101B-B826-00DD01103DE1),           //
 CLSID_Lines.
            helpstring("Lines Class"),
            appobject
      ]
      coclass Lines
      {
            [default] interface IApplication;
                  interface IDispatch;
      }
 }
```

The file concludes with the description of the Lines Application object, as specified by the **appobject** attribute. The **default** attribute applies to the **IApplication** interface, indicating that this interface will be returned by default.

Supporting Multiple National Languages

Applications sometimes need to expose objects with names that differ across localized versions of the product. The names pose a problem for programming languages that need to access these objects, because late binding will be sensitive to the locale of the application. The **IDispatch** interface provides a range of solutions that vary in cost of implementation and quality of language support. All methods of the **IDispatch** interface that are potentially sensitive to language are passed a locale ID (LCID), which identifies the local language context.

The following are some of the approaches a class implementation can take:

- Accept any locale ID and use the same member names in all locales. This is acceptable if the exposed interface will typically be accessed only by very advanced users. For example, the member names for OLE interfaces will never be localized.
- Accept all locale IDs supported by all versions of the product. In this case, the implementation of **GetIDsOfNames** would need to interpret the passed array of names based on the given locale ID. This is the most acceptable solution because it allows users to write code in their natural language and run the code on any localized version of the application.
- Return an error (DISP_E_UNKNOWNLCID) from **GetIDsOfNames** if the caller's locale ID does not match the localized version of the class. This prevents users from being able to write late-bound code that runs on machines with different localized implementations of the class.
- Recognize the particular version's localized names, as well as one language that is recognized in all versions. For example, a French version might accept French and English names, where English is the language supported in all versions. Users who want to write code that runs in all countries would have to use English names.

To provide general language support, the application should check the locale ID before interpreting member names. Because **Invoke** is passed a locale ID, methods can properly interpret parameters whose meaning varies by locale. The following sections provide examples and guidelines for creating multilingual applications.

Implementing IDispatch for Multilingual Applications

When creating applications that will support multiple languages, you need to create separate type libraries for each supported language, as well as for versions of the **IDispatch** member functions that include dependencies for each language. In the example below, the Hello sample code has been modified to define locale IDs for both U.S. English and German.

Implementing the IDispatch Member Functions

The following example code from the Hello sample implements language-sensitive versions of **GetTypeInfoCount, GetIDsOfNames,** and **Invoke.** Note that **Invoke** does not check the locale ID, but merely passes it to **DispInvoke. GetTypeInfoCount** does not contain any language-specific information; however, **GetTypeInfo** does.

The **IDispatch** member functions must be implemented in such a way as to take into account any language-specific features. **DispInvoke** is passed only the U.S. English type information pointer.

```
STDMETHODIMP
CHello::GetTypeInfoCount(UINT FAR* pctinfo)
{
     *pctinfo = 1;
     return NOERROR;
}

STDMETHODIMP
CHello::GetTypeInfo(
           UINT itinfo,
           LCID lcid,
           ITypeInfo FAR* FAR* pptinfo)
{
     LPTYPEINFO ptinfo;
     *pptinfo = NULL;

     if(itinfo != 0)
           return ResultFromScode(DISP_E_BADINDEX);

     if(lcid == LOCALE_SYSTEM_DEFAULT || lcid == 0)
           lcid = GettSystemDefaultLCID();

     if(lcid == LOCALE_USER_DEFAULT)
           lcid = GetUserDefaultLCID();

     switch(lcid)
     {
           case LCID_GERMAN:
                 ptinfo = m_ptinfoGerman;
                 break;

           case LCID_ENGLISH:
                 ptinfo = m_ptinfoEnglish;
                 break;

           default:
                 return ResultFromScode(DISP_E_UNKNOWNLCID);
     }

     ptinfo->AddRef();
     *pptinfo = ptinfo;
     return NOERROR;
}
```

```
STDMETHODIMP
CHello::GetIDsOfNames(
            REFIID riid,
            OLECHAR FAR* FAR* rgszNames,
            UINT cNames,
            LCID lcid,
            DISPID FAR* rgdispid)
{
      LPTYPEINFO ptinfo;

      if(lcid == LOCALE_SYSTEM_DEFAULT || lcid == 0)
            lcid = GetSystemDeraultLCID();

      if(lcid == LOCALE_USER_DEFAULT)
            lcid = GetUserDefaultLCID();

      switch(lcid)
      {
            case LCID_GERMAN:
                  ptinfo = m_ptinfoGerman;
                  break;

            case LCID_ENGLISH:
                  ptinfo = m_ptinfoEnglish;
                  break;

            default:
                  return ResultFromScode(DISP_E_UNKNOWNLCID);
      }
      return DispGetIDsOfNames(ptinfo, rgszNames, cNames, rgdispid);
}

STDMETHODIMP
CHello::Invoke(
      DISPID dispidMember,
      REFIID riid,
      LCID lcid,
      WORD wFlags,
      DISPPARAMS FAR* pdispparams,
      VARIANT FAR* pvarResult,
      EXCEPINFO FAR* pexcepinfo,
      UINT FAR* puArgErr)
{
      return DispInvoke(
      this, m_ptinfoEnglish,
      dispidMember, wFlags, pdispparams,
      pvarResult, pexcepinfo, puArgErr);
}

}
```

Creating Separate Type Libraries

For each supported language, write and register a separate type library. The type libraries use the same dispatch IDs and globally unique identifers, but you should localize names and Help strings based on the language. You must also define the locale IDs for the supported languages.

The following registration file example includes entries for U.S. English and German.

```
// Type library registration information.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0 =Hello
2.0 Type Library
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0\HELPDIR
=
// U.S. English.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0\409\
win16 = helloeng.tlb
// German.
HKEY_CLASSES_ROOT\TypeLib\{F37C8060-4AD5-101B-B826-00DD01103DE1}\2.0\407\
win16 = helloger.tlb
```

Defining the Locale IDs

Refer to the next section to obtain the language IDs for the supported languages.

```
// Locale IDs for the languages that are supported.
#define LCID_ENGLISH MAKELCID(MAKELANGID(0x09, 0x01))
#define LCID_GERMAN  MAKELCID(MAKELANGID(0x07, 0x01))
```

Using the example code from the Hello sample, define member variables that can be used to contain U.S. English and German type information.

```
class FAR CHello : public IHello
{
public:
     :

private:
     LPTYPEINFO m_ptinfoEnglish;   // English type information of Hello
                                                application interface.
     LPTYPEINFO m_ptinfoGerman;    // German type information of Hello
                                                        application
interface.
     :
};
```

Loading Type information

The following example uses the Hello sample code to illustrate the **LoadTypeInfo** function that loads locale-specific type library information when an object is created.

```
LoadTypeInfo(&phello->m_ptinfoEnglish, IID_IHello, LCID_ENGLISH);
LoadTypeInfo(&phello->m_ptinfoGerman, IID_IHello, LCID_GERMAN);

// LoadTypeInfo - Gets type information of an object's interface from
// the type library.
//
// Parameters:
// ppunkStdDispatch - Returns type information.
// clsid    - Interface ID of object in type library.
// lcid - Locale ID of type information to be loaded.
//
// Return Value:
// HRESULT
//
//
HRESULT LoadTypeInfo(ITypeInfo FAR* FAR* pptinfo, REFCLSID clsid,
LCID lcid)
{
     HRESULT hr;
     LPTYPELIB ptlib = NULL;
     LPTYPEINFO ptinfo = NULL;

     *pptinfo = NULL;

     // Load type library.
     hr = LoadRegTypeLib(LIBID_Hello, 2, 0, lcid, &ptlib);
     if (FAILED(hr))
           return hr;

     // Get type information for interface of the object.
     hr = ptlib->GetTypeInfoOfGuid(clsid, &ptinfo);
     if (FAILED(hr))
     {
           ptlib->Release();
           return hr;
     }

     ptlib->Release();
     *pptinfo = ptinfo;
     return NOERROR;
}
```

Interpreting Arguments and Strings Based on the Locale ID

Some methods or properties need to interpret arguments based on the locale ID (LCID). These methods or properties can require that an LCID be passed as an argument. Therefore, properties should be designed to have an LCID parameter.

The following example code of an object description language file implements a property that takes a locale ID.

```
[
      uuid(83219430-CB36-11cd-B774-00DD01103DE1),
      helpstring("Bank Account object."),
      oleautomation,
      dual
      ]
      interface IBankAccount : IDispatch
      {
            [propget, helpstring("Returns account balance formatted for the
                  country described by localeID.")]
            HRESULT CheckingBalance([in, lcid] long localeID, [out, retval]
            BSTR* retval);
            :
      }
```

In this example, **get_CheckingBalance** returns a currency string that contains the balance in the checking account. The currency string should be correctly formatted depending on the locale that is passed in. **ConvertCurrency** is a private function that converts the checking balance to the currency of the country described by the locale ID. The string form of converted currency is placed in **m_szBalance**. **GetCurrencyFormat** is a 32-bit Windows function that formats a currency string for the given locale.

The following represents the information contained in the header file:

```
class FAR CBankAccount : public IBankAccount
{
      public:
      // IUnknown methods.
            :

      // IDispatch methods.
            :

      // IBankAccount methods.
      STDMETHOD(get_CheckingBalance)(long llcid, BSTR FAR* pbstr);
            :
}
```

The following represents the .cpp file:

```
STDMETHODIMP
CBankAccount::get_CheckingBalance(long llcid, BSTR FAR* pbstr)
{
      TCHAR ach[100];
      ConvertCurrency(llcid);
            GetCurrencyFormat(llcid, 0, m_szBalance, NULL, ach,
            sizeof(ach));
```

```
            *pbstr = SysAllocString(ach);             // Return currency
string
     // formatted according to
                                                                    // locale
ID.

            return NOERROR;
}
```

The locale ID is commonly used to parse strings that contain locale-dependent information. For example, a function that takes a string such as "6/11/96" needs the locale ID to determine whether the month is June (6) or November (11). You should not use the locale ID for output strings, including error strings. These strings should always be displayed in the current system language.

Locale, Language, and Sublanguage IDs

The following macro is defined for creating locale IDs (Winnt.h for 32-bit systems; Olenls.h for 16-bit systems):

```
//*
// LCID creation/extraction macros:
//
// MAKELCID - construct locale ID from language ID and
// country code.
//
#define MAKELCID(l)     ((DWORD)(((WORD)(l)) | (((DWORD)((WORD)(0))) << 16)))
```

There are two predefined locale ID values. The system default locale is LOCALE_SYSTEM_DEFAULT, and the current user's locale is LOCALE_USER_DEFAULT.

Another macro constructs a language ID:

```
//
// Language ID creation/extraction macros:
//
// MAKELANGID - Construct language ID from primary language ID and
// sublanguage ID
//
#define MAKELANGID(p, s)                 ((((USHORT)(s)) << 10) | (USHORT)
(p))
```

The following three combinations of primary language ID and sublanguage ID have special meanings:

PRIMARYLANGID	SUBLANGID	Result
LANG_NEUTRAL	SUBLANG_NEUTRAL	Language neutral
LANG_NEUTRAL	SUBLANG_SYS_DEFAULT	System default language
LANG_NEUTRAL	SUBLANG_DEFAULT	User default language

For primary language IDs, the range 0x200 to 0x3ff is user definable. The range 0x000 to 0x1ff is reserved for system use. For sublanguage IDs, the range 0x20 to 0x3f is user definable. The range 0x00 to 0x1f is reserved for system use.

Language Tables

The following table lists the primary language IDs supported by Automation. For more information about national language support for 16-bit Windows systems, refer to Appendix A, "National Language Support Functions." For information about 32-bit Windows systems, see your operating system documentation.

Language	PRIMARYLANGID
Neutral	0x00
Chinese	0x04
Czech	0x05
Danish	0x06
Dutch	0x13
English	0x09
Finnish	0x0b
French	0x0c
German	0x07
Greek	0x08
Hungarian	0x0e
Icelandic	0x0F
Italian	0x10
Japanese	0x11
Korean	0x12
Norwegian	0x14
Polish	0x15
Portuguese	0x16
Russian	0x19
Slovak	0x1b
Spanish	0x0a
Swedish	0x1d
Turkish	0x1F

The following table lists the sublanguage IDs supported by Automation. For more information about national language support for 16-bit systems, refer to Appendix A, "National Language Support Functions." For information about 32-bit Windows systems, see your operating system documentation.

Sublanguage	SUBLANGID
Neutral	0x00
Default	0x01
System Default	0x02
Chinese (Simplified)	0x02
Chinese (Traditional)	0x01
Czech	0x01
Danish	0x01
Dutch	0x01
Dutch (Belgian)	0x02
English (U.S.)	0x01
English (UK)	0x02

English (Australian)	0x03
English (Canadian)	0x04
English (Irish)	0x06
English (New Zealand)	0x05
Finnish	0x01
French	0x01
French (Belgian)	0x02
French (Canadian)	0x03
French (Swiss)	0x04
German	0x01
German (Swiss)	0x02
German (Austrian)	0x03
Greek	0x01
Hungarian	0x01
Icelandic	0x01
Italian	0x01
Italian (Swiss)	0x02
Japanese	0x01
Korean	0x01
Norwegian (Bokmal)	0x01
Norwegian (Nynorsk)	0x02
Polish	0x01
Portuguese	0x02
Portuguese (Brazilian)	0x01
Russian	0x01
Slovak	0x01
Spanish (Castilian) (1)	0x01
Spanish (Mexican)	0x02
Spanish (Modern) (1)	0x03
Swedish	0x01
Turkish	0x01

1 The only difference between Spanish (Castilian) and Spanish (Modern) is the sort ordering. The LCType values are the same.

Accessing ActiveX™ Objects

To access exposed objects, you can create ActiveX clients using Visual Basic, Visual C++®, Microsoft Excel, and other applications and programming languages that support the Automation technology. This chapter discusses several strategies for accessing exposed objects.

- Creating scripts with Visual Basic
- Creating controllers that manipulate objects
- Creating type information browsers

Regardless of your strategy, an ActiveX client needs to follow these steps:

Quick Info To initialize and create the object

1. Initialize OLE.
2. Create an instance of the exposed object.

Quick Info To manipulate methods and properties

1. Get information about the object's methods and properties.
2. Invoke the methods and properties.

Quick Info To release OLE when the application or programming tool terminates

1. Revoke the active object.
2. Unitialize OLE.

Note Throughout this chapter, the file names of sample applications appear in parentheses before the sample code.

Creating Scripts Using Visual Basic

Visual Basic provides a complete programming environment for creating Windows applications with which you can manipulate the exposed ActiveX objects of other applications. Internally, Visual Basic fully supports Automation dual interfaces.

For the syntax and semantics of the Automation features, see the Visual Basic Help file, Vb.hlp. To see how the Visual Basic statements translate into ActiveX application programming interfaces (APIs), refer to Appendix C, "Information for Visual Basic Programmers."

> **Note** Visual Basic is not necessary to use Automation. It is presented here as an example of a programming tool that supports Automation and is convenient for packaging Automation scripts. Optionally, a different ActiveX client can be used for testing.

Exposed objects can be called directly from programs written with Visual Basic. The following figure shows how this was done for the sample program Hello.exe.

{ewc msdncd, EWGraphic, bsd23525 0 /a "SDK_03.BMP"}

Quick Info

To access an exposed object

1. Start Visual Basic. (Initialization and release of OLE is handled automatically by Visual Basic.)
2. To select the type library of the object, click **Tools** on the **References** menu.
3. Add code to declare a variable of the interface type. For example:

   ```
   Dim HelloObj As IHello
   ```

4. Add code in event procedures to create an instance of the object and to manipulate the object using its properties and methods. For example:

   ```
   Sub Form_Load ( )
      Set HelloObj = New Hello.Hello
   End Sub
   Sub SetVisible_Click ( )
      HelloObj.Visible = True
   End Sub
   ```

5. Click **Start** on the **Run** menu, and then trigger the event by clicking the form.

The following figure shows the interfaces you use when accessing exposed objects through Visual Basic.

{ewc msdncd, EWGraphic, bsd23525 1 /a "SDK_10.WMF"}

Accessing a Remote Object

With Visual Basic, accessing a remote object requires only that the program declare an object variable and assign the return of a **New** statement to the variable. The following is the syntax for the statements:

Dim *ObjectVar* **As** *InterfaceName*
Set *ObjectVar* **= New** *CoClassName*

The **Dim** statement declares a variable of an interface type. The **New** keyword creates an instance of an object. Used together, the two declare and create an instance of an ActiveX object. For example:

```
Dim MyLines As ILines
Set MyLines = New Lines.Lines
```

The **Dim** statement declares the object variable `MyLines` of the interface type **ILines.** The **Set** statement assigns a new object of the *component object class* (coclass) Lines to the variable `MyLines`**.** When you use the **Dim** statement to set a variable to an interface type, subsequent uses of the variable will execute faster than with the generic Object syntax.

The **New** keyword applies only to creating coclasses of interface or dispinterface types. To create other types of objects, the variable must be declared with the **Dim** statement, and the **CreateObject** function used as follows:

Set *ObjectVar* **= CreateObject(***ProgID***)**

CreateObject creates an ActiveX object, based on the specified programmatic ID (ProgID). The ProgID has the form:

AppName.*ObjectName*

The *AppName* is the name of the application, and the *ObjectName* identifies the type of object to create. For more information about ProgIDs, see "Registering the Application" in Chapter 2, "Exposing ActiveX Objects."

You can use the **GetObject** function to re-establish the reference to the most recently used object that corresponds to the *Filename* and *AppName.ObjectName* specification, as folllows:

Set *ObjectVar* **= GetObject("***Filename***",** *ProgID***)**

For example, if the ActiveX client needs an existing instance of an object instead of a new instance, it can use the **GetObject** function.

The Hello sample application, included in the Win32 Software Development Kit (SDK), displays a Hello message in response to a mouse click. You can add a simple form that accesses the Hello application's exposed object from another process.

Quick Info

To add a form

1. Start Visual Basic.
2. Click **Open Project** on the **File** menu.
3. In the dialog box, click Vb.mak in the Sample directory.
4. In the **Forms** box, click **View Form** to see the form, or click **View Code** to see the Visual Basic code.

```
'Module-level declarations
Dim HelloObj As IHello

Sub Form_load ( )
```

```
   Set HelloObj = New Hello.Hello
End Sub

Sub Invoke_SayHello_Method_Click ( )
   HelloObj.SayHello
End Sub

Sub Get_HelloMsg_Property_Click ()
   Debug.Print HelloObj.HelloMessage
End Sub

Sub Set_HelloMsg_Property_Click ( )
   HelloObj.HelloMessage = "Hello Universe"
End Sub

Sub SetVisible_Click ( )
   HelloObj.Visible = True
End Sub
```

The **Form_Load** subroutine creates the Hello Application object. Other subroutines manipulate the Hello Application object through the **Visible** and **HelloMessage** properties and the **SayHello** method.

To program an object in Visual Basic, you need its class name and the names and parameters of its properties and methods. For the Hello sample application, you need to know the exact names of the **SayHello** method, and the **Visible** and **HelloMsg** properties, along with the types of their arguments. This information is provided in the documentation for many objects, such as those exposed by Microsoft Excel. You can also get the information by viewing the object's type library with an object browser, such as the one included in Visual Basic. A sample browser, Browse, is provided in the Win32 SDK.

Creating an Invisible Object

In the preceding section, Visual Basic was used to access and program a form-based interface for the Hello sample. The Hello Application object was started as invisible, and was later displayed when its **Visible** property was set to **True**. Some objects are not visible, and some objects are never displayed to the user.

For example, a word-processing application may expose its spelling checker engine as an object. This object might support a method called **CheckWord** that takes a string as an argument. If the string is spelled correctly, the method returns **True**; otherwise, the method returns **False**. If the string is spelled incorrectly, it could be passed to another (hypothetical) method called **SuggestWord** that returns a suggestion for its correct spelling. The code might look something like this.

```
Sub CheckSpelling ()
     Dim ObjVar As New SpellChecker
     Dim MyWord, Result

     MyWord = "potatoe"

     ' Check the spelling.
     Result = ObjVar.CheckWord MyWord
     ' If False, get suggestion.
     If Not Result Then
           MyWord = ObjVar.SuggestWord MyWord
     End If
End Sub
```

In this example, the spelling checker is never displayed to the user. Its capabilities are exposed through the properties and methods of the spelling checker object.

As shown in the example, invisible objects can be created and referenced in the same way as any other type of object.

Activating an Object from a File

Many Automation applications let the user save objects in files. For example, a spreadsheet application that supports Worksheet objects lets the user save the worksheet in a file. The same application may also support a Chart object that the user can save in a file.

To activate an object from a file, first declare an object variable, and then call the **GetObject** function using the following syntax:

GetObject (*filename***[, *ProgID*])**

The *filename* argument is a string containing the full path and name of the file to be activated. For example, an application named SpdSheet.exe creates an object that was saved in a file named Revenue.spd. The following code invokes Spdsheet.exe, loads the file Revenue.spd, and assigns Revenue.spd to an object variable:

```
Dim Ss As Spreadsheet
Set Ss = GetObject("C:\Accounts\Revenue.spd")
```

If the *filename* argument is omitted, then **GetObject** returns the currently active object of the specified ProgID. For example:

```
Set Ss = GetObject (,"SpdSheet.Application")
```

If there is no active object of the class SpdSheet.Application, an error will occur.

In addition to activating an entire file, some applications let you activate part of a file. To activate part of a file, add an exclamation point (!) or a backslash (\) to the end of the file name, followed by a string that identifies the part of the file you want to activate. For information on how to create this string, refer to the object's documentation.

For example, if SpdSheet.exe is a spreadsheet application that uses R1C1 syntax, the following code could be used to activate a range of cells within Revenue.spd:

```
Set Ss = GetObject("C:\Accounts\Revenue.spd!R1C1:R10C20")
```

These examples invoke an application and activate an object. In these examples, the application name (SpdSheet.exe) is never specified. When **GetObject** is used to activate an object, the registry files determine the application to invoke and the object to activate based on the file name or ProgID that is provided. If a ProgID is not provided, Automation activates the default object of the specified file.

Some ActiveX components, however, support more than one class of object. Suppose the spreadsheet file, Revenue.spd, supports three different classes of objects: an Application object, a Worksheet object, and a Toolbar object, all of which are part of the same file. To specify which object to activate, an argument must be supplied for the optional ProgID parameter. For example:

```
Set Ss = GetObject("C:\Revenue.spd", "SpdSheet.Toolbar")
```

This statement activates the SpdSheet.Toolbar object in the file Revenue.spd.

Accessing Linked and Embedded Objects

Some applications that supply objects support linking and embedding as well as Automation. Using the ActiveX control (Msole2.vbx) from the OLE toolkit, you can create and display embedded objects in a Visual Basic application. If the objects also support Automation, you can access their properties and methods by using the **Object** property. The **Object** property returns the object in the ActiveX control. This property refers to an ActiveX object in the same way an object variable created with the functions **New**, **CreateObject**, or **GetObject** refers to the object.

For example, an ActiveX control named Ole1 contains an object that supports Automation. This object has an **Insert** method, a **Select** method, and a **Bold** property. In this case, the following code could be written to manipulate the ActiveX control's object:

```
' Insert text in the object.
Ole1.Object.Insert "Hello, world."
' Select the text.
Ole1.Object.Select
' Format the text as bold.
Ole1.Object.Bold = True
```

Manipulating Objects

Once a variable has been created that references an ActiveX object, the object can be manipulated in the same way as any other Visual Basic object. To get and set an object's properties, or to perform an object's methods, use the *object.property* or *object.method* syntax. Multiple objects, properties, and methods can be included on the same line of code using the following syntax:

```
ObjVar.Cell(1,1).FontBold = True
```

Accessing the Properties of an Object

To assign a value to a property of an object, put the object variable and property name on the left side of an assignment, and the desired property setting on the right side. For example:

```
Dim ObjVar As IMyInterface
Dim RowPos, ColPos
Set ObjVar = New MyObject

ObjVar.Text = "Hello, world"
ObjVar.Cell(RowPos, ColPos) = "This property accepts two arguments."
' Sets the font for ObjVar.Selection.
ObjVar.Selection.Font = 12
```

Property values can also be retrieved from an object:

```
Dim X As Object

X = ObjVar.Text
X = ObjVar.Range(12, 32)
```

Invoking Methods

In addition to getting and setting properties, an object can be manipulated using the methods it supports. Some methods may return a value, as in the following example:

```
X = ObjVar.Calculate(1,2,3)
```

Methods that do not return a value behave like subroutines. For example:

```
' This method requires two arguments.
ObjVar.Move XPos, YPos
```

If such a method is assigned to a variable, an error will occur.

Creating Applications and Tools That Access Objects

Automation provides interfaces for accessing exposed objects from an application or programming tool written in C or C++. The following sections show C++ code that uses the same type of access method as the Visual Basic code, which is described earlier in this chapter. Although the process is more complex with C++ than with Visual Basic, the approach is similar. This section shows the minimum code necessary to access and manipulate a remote object.

You can use the **IDispatch** interface to access ActiveX objects, or you can access objects directly through the VTBL. Because VTBL references can be bound at compile time, VTBL access is generally faster than through **IDispatch**. Whenever possible, use the **ITypeInfo** interface to get information about an object, including the VTBL addresses of the object's members. Then, use this information to access ActiveX objects through the VTBL.

To create compilers and other programming tools that use information from type libraries, use the **ITypeComp** interface. This interface binds to exposed objects at compile time. For details on **ITypeComp**, see Chapter 9, "Type Description Interfaces."

Accessing Members Through VTBLs

For objects that have dual interfaces, the first seven members of the VTBL are the members of **IUnknown** and **IDispatch**, and the subsequent members are standard COM entries for the interface's member functions. You can call these entries directly from C++.

Quick Info

To access a method or property through the VTBL

1. Initialize OLE.
2. Create an instance of the exposed object.
3. Manipulate the properties and methods of the object.
4. Uninitialize OLE.

The code sample that follows shows how to access a property of the Hello object. Error handling has been omitted for brevity (Hello.vbp).

```
HRESULT hr;
CLSID clsid;                        // Class ID of Hello object.
LPUNKNOWN punk = NULL;              // Unknown of Hello object.
IHello* phello = NULL;              // IHello interface of Hello object.

// Initialize OLE.
hr = OleInitialize(NULL);

// Retrieve CLSID from the ProgID for Hello.
hr = CLSIDFromProgID("Hello.Application", &clsid);

// Create an instance of the Hello object and ask for its
// IDispatch interface.
hr = CoCreateInstance(clsid, NULL, CLSCTX_SERVER,
                      IID_IUnknown, (void FAR* FAR*)&punk);

hr = punk->QueryInterface(IID_IHello, (void FAR* FAR*)&pHello);

punk->Release();                // Release when no longer needed.

hr = pHello->put_Visible (TRUE);

// Additional code to work with other methods and properties.
// .
// .
// .

OleUninitialize();
```

The example initializes OLE, and then calls the **CLSIDFromProgID** function to obtain the CLSID for the Hello application. With the CLSID, the example can call **CoCreateInstance** to create an instance of the Hello Application object. **CoCreateInstance** returns a pointer to the object's **IUnknown** interface (punk), and this, in turn, is used to call **QueryInterface** to get pHello, a pointer to the IID_IHello dual interface. The punk is no longer needed, so the example releases it. The example then sets the value of the **Visible** property to **True**.

If the function returns an error HRESULT, you can get detailed, contextual information through the **IErrorInfo** interface. For details, see Chapter 11, "Error Handling Interfaces."

Accessing Members Through IDispatch

To bind to exposed objects at run time, use the **IDispatch** interface.

Quick Info

To create an ActiveX client using IDispatch

1. Initialize OLE.
2. Create an instance of the object you want to access. The object's ActiveX component creates the object.
3. Obtain a reference to the object's **IDispatch** interface (if it has implemented one).
4. Manipulate the object through the methods and properties exposed in its **IDispatch** interface.
5. Terminate the object by invoking the appropriate method in its **IDispatch** interface, or by releasing all references to the object.
6. Uninitialize OLE.

The following table shows the minimum set of functions necessary to manipulate a remote object.

Function	Purpose	Interface
OleInitialize	Initializes OLE.	OLE API function
CoCreateInstance	Creates an instance of the class represented by the specified CLSID, and returns a pointer to the object's **IUnknown** interface.	Component object API function
QueryInterface	Checks whether **IDispatch** has been implemented for the object. If so, returns a pointer to the **IDispatch** implementation.	**IUnknown**
GetIDsOfNames	Returns DISPIDs for properties and methods and their parameters.	**IDispatch**
Invoke	Invokes a method, or sets or gets a property of the remote object.	**IDispatch**
Release	Decrements the reference count for an **IUnknown** or **IDispatch** object.	**IUnknown**
OleUninitialize	Uninitializes OLE.	OLE API function

The code that follows is extracted from a generalized Windows-based ActiveX client. The controller relies on helper functions provided in the file Invhelp.cpp, which is available in the Browse directory of the samples. Error checking is omitted to save space, but would normally be used where an HRESULT is returned.

The two functions that follow initialize OLE, and then create an instance of an object and get a pointer to the object's **IDispatch** interface (Invhelp.cpp):

```
BOOL InitOle(void)
{
    if(OleInitialize(NULL) != 0)
        return FALSE;
```

```
    return TRUE;
}
HRESULT CreateObject(LPSTR pszProgID, IDispatch FAR* FAR* ppdisp)
{
    CLSID clsid;                  // CLSID of ActiveX object.
    HRESULT hr;
    LPUNKNOWN punk = NULL;        // IUnknown of ActiveX object.
    LPDISPATCH pdisp = NULL;      // IDispatch of ActiveX object.

    *ppdisp = NULL;

    // Retrieve CLSID from the ProgID that the user specified.
    hr = CLSIDFromProgID(pszProgID, &clsid);
    if (FAILED(hr))
        goto error;

    // Create an instance of the ActiveX object and ask for the
    // IDispatch interface.
    hr = CoCreateInstance(clsid, NULL, CLSCTX_SERVER,
                          IID_IUnknown, (void FAR* FAR*)&punk);
    if (FAILED(hr))
        goto error;

    hr = punk->QueryInterface(IID_IDispatch, (void FAR* FAR*)&pdisp);
    if (FAILED(hr))
        goto error;

    *ppdisp = pdisp;
    punk->Release();
    return NOERROR;

error:
    if (punk) punk->Release();
    if (pdisp) pdisp->Release();
    return hr;
}
```

The **CreateObject** function is passed a **ProgID** and returns a pointer to the **IDispatch** implementation of the specified object. **CreateObject** calls the OLE API **CLSIDFromProgID** to get the CLSID that corresponds to the requested object, and then passes the CLSID to **CoCreateInstance** to create an instance of the object and get a pointer to the object's **IUnknown** interface. (The **CLSIDFromProgID** function is described in the *Microsoft OLE Programmer's Guide and Reference.)* With this pointer, **CreateObject** calls **IUnknown::QueryInterface,** specifying IID_IDispatch, to get a pointer to the object's **IDispatch** interface.

```
HRESULT FAR
Invoke(LPDISPATCH pdisp,
    WORD wFlags,
    LPVARIANT pvRet,
    EXCEPINFO FAR* pexcepinfo,
    UINT FAR* pnArgErr,
    LPSTR pszName,
    char *pszFmt,
    ...)
{
```

```
va_list argList;
va_start(argList, pszFmt);
DISPID dispid;
HRESULT hr;
VARIANTARG* pvarg = NULL;

if (pdisp == NULL)
    return ResultFromScode(E_INVALIDARG);

// Get DISPID of property/method.
hr = pdisp->GetIDsOfNames(IID_NULL, &pszName, 1,
    LOCALE_SYSTEM_DEFAULT, &dispid);
if(FAILED(hr))
    return hr;

DISPPARAMS dispparams;
_fmemset(&dispparams, 0, sizeof dispparams);

// Determine number of arguments.
if (pszFmt != NULL)
    CountArgsInFormat(pszFmt, &dispparams.cArgs);

// Property puts have a named argument that represents the value
// being assigned to the property.
DISPID dispidNamed = DISPID_PROPERTYPUT;
if (wFlags & DISPATCH_PROPERTYPUT)
{
    if (dispparams.cArgs == 0)
        return ResultFromScode(E_INVALIDARG);
    dispparams.cNamedArgs = 1;
    dispparams.rgdispidNamedArgs = &dispidNamed;
}
if (dispparams.cArgs != 0)
{
    // Allocate memory for all VARIANTARG parameters.
    pvarg = new VARIANTARG[dispparams.cArgs];
    if(pvarg == NULL)
        return ResultFromScode(E_OUTOFMEMORY);
    dispparams.rgvarg = pvarg;
    _fmemset(pvarg, 0, sizeof(VARIANTARG) * dispparams.cArgs);

    // Get ready to traverse the vararg list.
    LPSTR psz = pszFmt;
    pvarg += dispparams.cArgs - 1;   // Params go in opposite order.

    while (psz = GetNextVarType(psz, &pvarg->vt))
    {
        if (pvarg < dispparams.rgvarg)
        {
            hr = ResultFromScode(E_INVALIDARG);
            goto cleanup;
        }
        switch (pvarg->vt)
        {
        case VT_I2:
```

```
                V_I2(pvarg) = va_arg(argList, short);
                break;
            case VT_I4:
                V_I4(pvarg) = va_arg(argList, long);
                break;
            // Additional cases omitted to save space.
            default:
                {
                    hr = ResultFromScode(E_INVALIDARG);
                    goto cleanup;
                }
                break;
            }
            --pvarg;             // Get ready to fill next argument.
        } //while
    } //if

    // Initialize return variant, in case caller forgot. Caller can pass
    // Null if no return value is expected.
    if (pvRet)
        VariantInit(pvRet);
    // Make the call.
    hr = pdisp->Invoke(dispid, IID_NULL, LOCALE_SYSTEM_DEFAULT, wFlags,
        &dispparams, pvRet, pexcepinfo, pnArgErr);

cleanup:
    // Clean up any arguments that need it.
    if (dispparams.cArgs != 0)
    {
        VARIANTARG FAR* pvarg = dispparams.rgvarg;
        UINT cArgs = dispparams.cArgs;
        while (cArgs--)
        {
            switch (pvarg->vt)
            {
            case VT_BSTR:
                VariantClear(pvarg);
                break;
            }
            ++pvarg;
        }
    }
    delete dispparams.rgvarg;
    va_end(argList);
    return hr;
}
```

In this example, the **Invoke** function is a general-purpose function that calls **IDispatch::Invoke** to invoke a property or method of an ActiveX object. As arguments, it accepts the object's **IDispatch** implementation, the name of the member to invoke, flags that control the invocation, and a variable list of the member's arguments. It can be found in the Browse sample in the file Invelp.ccp.

Using the object's **IDispatch** implementation and the name of the member, it calls **GetIDsOfNames** to get the dispatch ID (DISPID) of the requested member. The member's DISPID must be used later, in the call to **IDispatch::Invoke**.

The invocation flags specify whether a method, PROPERTYPUT, or PROPERTYGET function is being invoked. The helper function simply passes these flags directly to **IDispatch::Invoke**.

The helper function next fills in the DISPPARAMS structure with the parameters of the member. DISPPARAMS structures have the following form:

```
typedef struct FARSTRUCT tagDISPPARAMS{
    VARIANTARG FAR* rgvarg;           // Array of arguments.
    DISPID FAR* rgdispidNamedArgs;    // DISPIDs of named arguments.
    UINT cArgs;                       // Number of arguments.
    UINT cNamedArgs;                  // Number of named arguments.
} DISPPARAMS;
```

The *rgvarg* field is a pointer to an array of VARIANTARG structures. Each element of the array specifies an argument, whose position in the array corresponds to its position in the parameter list of the method definition. The *cArgs* field specifies the total number of arguments, and the *cNamedArgs* field specifies the number of named arguments.

For methods and property get functions, all arguments can be accessed as positional, or they can be accessed as named arguments. Property put functions have a named argument that is the new value for the property. The DISPID of this argument is DISPID_PROPERTYPUT.

To build the *rgvarg* array, the **Invoke** helper function retrieves the parameter values and types from its own argument list, and constructs a VARIANTARG structure for each one. (For a description of the format string that specifies the types of the parameters, see the file Invhelp.cpp.) Parameters are put in the array in reverse order, so that the last parameter is in *rgvarg*[0], and so forth. Although VARIANTARG has the following five fields, only the first and fifth are used.

```
typedef struct FARSTRUCT tagVARIANT VARIANTARG;

struct FARSTRUCT tagVARIANT{
    VARTYPE vt;
    unsigned short wReserved1;
    unsigned short wReserved2;
    unsigned short wReserved3;
    union {
        short        iVal;        /* VT_I2        */
.
.    // The rest of this union specifies numerous other types.
.

    };
} VARIANTARG;
```

The first field contains the argument's type, and the fifth contains its value. To pass a long integer, for example, the *vt* and *iVal* fields of the VARIANTARG structure would be filled with VT_I4 (long integer) and the actual value of the long integer.

In addition, for property put functions, the first element of the *rgdispidNamedArgs* array must contain DISPID_PROPERTYPUT.

After filling the DISPPARAMS structure, the **Invoke** helper function initializes `pvRet`, a variant in which **IDispatch::Invoke** returns a value from the method or property. The following is the actual call to **IDispatch::Invoke**:

```
hr = pdisp->Invoke(dispid, IID_NULL, LOCALE_SYSTEM_DEFAULT, wFlags,
        &dispparams, pvRet, pexcepinfo, pnArgErr);
```

The variable pdisp is a pointer to the object's **IDispatch** interface. DISPID indicates the method or property being invoked. The value IID_NULL must be specified for all **IDispatch::Invoke** calls, and LOCALE_SYSTEM_DEFAULT is a constant denoting the default locale ID (LCID) for this system. In the final two arguments, pexcepinfo and pnArgErr, **IDispatch::Invoke** can return error information.

If the invoked member has defined an exception handler, it returns exception information in pexcepinfo. If certain errors occur in the argument vector, pnArgErr points to the errant argument. The function return value hr is an HRESULT that indicates success or various types of failure.

For more information, including how to pass optional arguments, see "IDispatch::Invoke" in Chapter 5, "Dispatch Interface and API Functions."

Creating Type Information Browsers

Type information browsers let users scan type libraries to determine what types of objects are available. The following figure shows the interfaces you can use when creating compilers or browsers that access type libraries.

{ewc msdncd, EWGraphic, bsd23525 2 /a "SDK_01.WMF"}

The Browse and BrowseH samples show how a browser might access a type library. The Browse sample is a Windows-based browser that presents a dialog box from which type information items can be selected to display. This example function prompts for the name of the type library, opens the library, and gathers and displays information.

Standard Objects and Naming Guidelines

This chapter describes the standard ActiveX objects, and discusses naming guidelines for creating objects that are unique to applications, especially user-interactive applications that support a multiple-document interface (MDI). If an ActiveX object is not user-interactive or supports only a single-document interface (SDI), the standards and guidelines should be adapted as appropriate.

- *Standard objects* comprise a set of objects defined by Automation. You can use them as appropriate to your application. The objects described in this chapter are oriented toward document-based, user-interactive applications. Other applications (such as noninteractive database servers) may have different requirements.
- *Naming guidelines* are recommendations meant to improve consistency across applications.

This chapter also provides examples in a hypothetical syntax derived from Visual Basic. The standards and guidelines are subject to change.

Using Standard Objects

The following table lists the Automation standard objects. Although none of these objects are required, user-interactive applications with subordinate objects should include an Application object.

Object name	Description
Application	Top-level object. Provides a standard way for ActiveX clients to retrieve and navigate an application's subordinate objects.
Document	Provides a way to open, print, change, and save an application document.
Documents	Provides a way to iterate over and select open documents in multiple-document interface (MDI) applications.
Font	Describes fonts that are used to display or print text.

The following illustration shows how the standard objects fit into the organization of objects provided by an application.

{ewc msdncd, EWGraphic, bsd23526 0 /a "SDK_01.WMF"}

The following sections describe the standard properties and methods for all objects, all collection objects, and each of the standard objects. These sections list the standard methods and properties for each object, as well as the standard arguments for those properties and methods.

Note You can define additional application-specific properties and methods for each object. You can also provide additional optional arguments for any of the listed properties or methods; however, the optional arguments should follow the standard arguments in a positional argument list.

Object Properties

All objects, including the Application object and collection objects, must provide the following properties:

Property name	Return type	Description
Application	VT_DISPATCH	Returns the Application object; read only.
Parent	VT_DISPATCH	Returns the creator of the object; read only.

Note The **Application** and **Parent** properties of the Application object return the Application object.

Collection Object Properties

A collection provides a set of objects over which iteration can be performed. All collection objects must provide the following properties:

Property name	Return type	Description
Count	VT_I4	Returns the number of items in the collection; read only. Required.
_NewEnum	VT_DISPATCH	A special property that returns an enumerator object that implements **IEnumVARIANT**. Required.

Collection Methods

Methods for collections are described in the following table. The **Item** method is required; other methods are optional.

Method name	Return type	Description
Add	VT_DISPATCH or VT_EMPTY	Adds an item to a collection. Returns VT_DISPATCH if object is created (object cannot exist outside the collection) or VT_EMPTY if no object is created (object can exist outside the collection).
Item	Varies with type of collection	Returns the indicated item in the collection. Required. The **Item** method may take one or more arguments to indicate the element within the collection to return. This method is the default member (DISPID_VALUE) for the collection object.
Remove	VT_EMPTY	Removes an item from a collection. Uses indexing arguments in the same way as the **Item** method.

All collection objects must provide at least one form of indexing through the **Item** method. The dispatch ID of the **Item** method is DISPID_VALUE. Because it is the default member, it can be used in the following form:

```
ThirdDef = MyWords(3).Definition   ' Equivalent to
                                                    '
MyWords.Item(3).Definition
```

The **Item** method takes one or more arguments to indicate the index. Indexes can be numbers, strings, or other types. For example:

```
DogDef = MyWords("dog").Definition
```

Important Within the application's type library, the **_NewEnum** property has a special dispatch ID: DISPID_NEWENUM. The name **_NewEnum** should not be localized.

The **Add** method may take one or more arguments. For example, if **MyWord** is an object with the properties **Letters** and **Definition**:

```
Dim MyWord As New Word
Dim MyDictionary as Words
MyWord = "dog"
MyWord.Letters = "Dog"
MyWord.Definition = "My best friend."
MyDictionary.Add MyWord
MyDictionary.Remove("Dog")
```

For more information about creating collection objects, see Chapter 2, "Exposing ActiveX objects."

Kinds of Collections

The standard for collections lets you describe two kinds of collections, depending on whether it makes sense for the collected objects to exist outside the collection.

In some cases, it is not logical for an object to exist independently of its collection. For example, an application's Documents collection contains all Document objects currently open. Opening a document means adding it to the collection, and closing the document means removing it from the collection. All open documents are part of the collection. The application cannot have open documents that are not part of the collection. The relationship between the collection and the members of the collection can be shown in the following ways:

- **Documents.Add** creates an object (an open document) and adds it to the collection. Because an object is created, a reference to it is returned.

  ```
  Set MyDoc = Documents.Add
  ```

- **Document.Close** removes an object from the collection.

  ```
  Set SomeDoc = Documents(3)
  SomeDoc.Close
  ```

In other cases, it is logical for the objects to exist outside the collection. For example, a Mail application might have Name objects, and many collections of these Name objects. Each Name object would have a user's e-mail name, full name, and possibly other information. The e-mail name and full name would likely be properties named **EmailName** and **FullName.**

Additionally, the application might have the following collections of Name objects.

- A collection for the "To" list on each piece of e-mail.
- A collection of the names of the people to whom a user has sent e-mail.

The collections of Name objects could be indexed by using either **EmailName** or **FullName**.

For these collections, the **Add** method does not create an object because the object already exists. Therefore, the **Add** method should take an object as an argument, and should not return a value.

Assuming the existence of two collections (**AddressBook** and **ToList**), a user might execute the following code to add a Name object to the **ToList** collection:

```
Dim Message as Object
Dim AddressBook as Object
Dim NameRef as Object
.
.
.

Set NameRef = AddressBook.Names("Fred Funk")
Message.ToList.Add   NameRef
```

The Name object already exists and is contained in the **AddressBook** collection. The first line of code obtains a reference to the Name object for "Fred Funk" and points to **NameRef**. The second line of code adds a reference to the object to the **ToList** collection. No new object is created, so no reference is returned from the **Add** method.

Unlike the relationship between Documents and Document, there is no way for the collected object (the Name) to know how to remove itself from the collections in which it is contained. To remove an item from a collection, use the **Remove** method, as follows:

```
Message.ToList.Remove("Fred Funk")
```

This line of code removes the Name object that has the **FullName** "Fred Funk." The "Fred Funk" object may exist in other collections, but they will be unaffected.

Using the Application Object in a Type Library

If you use a type library, the Application object should be the object that has the **appobj** attribute. Because some ActiveX clients use the type information to allow unqualified access to the Application object's members, it is important to avoid overloading the Application object with too many members.

The Application object should have the properties listed in the following table. The **Application, FullName, Name, Parent,** and **Visible** properties are required; other properties are optional.

Property name	Return type	Description
ActiveDocument	VT_DISPATCH, VT_EMPTY	Returns the active document object or VT_EMPTY if none; read only.
Application	VT_DISPATCH	Returns the Application object; read only. Required.
Caption	VT_BSTR	Sets or returns the title of the application window; read/write. Setting the Caption to VT_EMPTY returns control to the application.
DefaultFilePath	VT_BSTR	Sets or returns the default path specification used by the application for opening files; read/write.
Documents	VT_DISPATCH	Returns a collection object for the open documents; read only.
FullName	VT_BSTR	Returns the file specification for the application, including path; read only. For example, C:\Drawdir\Scribble. Required.
Height	VT_R4	Sets or returns the distance between the top and bottom edge of the main application window; read/write.
Interactive	VT_BOOL	Sets or returns True if the application accepts actions from the user, otherwise False; read/write.
Left	VT_R4	Sets or returns the distance between the left edge of the physical screen and the main application window; read/write.
Name	VT_BSTR	Returns the name of the application, such as "Microsoft Excel"; read only. The **Name** property is the default member (DISPID_VALUE) for the Application object. Required.
Parent	VT_DISPATCH	Returns the Application object; read only. Required.
Path	VT_BSTR	Returns the path specification for the application's executable file; read only. For example, C:\Drawdir if the .exe file is C:\Drawdir\Scribble.exe.
StatusBar	VT_BSTR	Sets or returns the text displayed in the status bar; read/write.
Top	VT_R4	Sets or returns the distance between

		the top edge of the physical screen and main application window; read/write.
Visible	VT_BOOL	Sets or returns whether the application is visible to the user; read/write. The default is False when the application is started with the **/Automation** command-line switch. Required.
Width	VT_R4	Sets or returns the distance between the left and right edges of the main application window; read/write.

The Application object should have the following methods. The **Quit** method is required; other methods are optional.

Method name	**Return type**	**Description**
Help	VT_EMPTY	Displays online Help. May take three optional arguments: *helpfile* (VT_BSTR), *helpcontextID* (VT_I4), and *helpstring* (VT_BSTR). The *helpfile* argument specifies the Help file to display; if omitted, the main Help file for the application is displayed. The *helpcontextID* and *helpstring* arguments specify a Help context to display; only one of them can be supplied. If both are omitted, the default Help topic is displayed.
Quit	VT_EMPTY	Exits the application and closes all open documents. Required.
Repeat	VT_EMPTY	Repeats the previous action in the user interface.
Undo	VT_EMPTY	Reverses the previous action in the user interface.

Document Object Properties

If the application is document based, it should provide a Document object named Document. Use a different name only if Document is inappropriate (for example, if the application uses highly technical or otherwise specialized terminology within its user interface).

The Document object should have the properties listed in the table that follows. The properties **Application, FullName, Name, Parent, Path,** and **Saved** are required; other properties are optional.

Property name	Return type	Description
Application	VT_DISPATCH	Returns the Application object; read only. Required.
Author	VT_BSTR	Sets or returns summary information about the document's author; read/write.
Comments	VT_BSTR	Sets or returns summary information comments for the document; read/write.
FullName	VT_BSTR	Returns the file specification of the document, including the path; read only. Required.
Keywords	VT_BSTR	Sets or returns summary information keywords associated with the document; read/write.
Name	VT_BSTR	Returns the file name of the document, not including the file's path specification; read only.
Parent	VT_DISPATCH	Returns the **Parent** property of the Document object; read only. Required.
Path	VT_BSTR	Returns the path specification for the document, not including the file name or file name extension; read only. Required.
ReadOnly	VT_BOOL	Returns True if the file is read only, otherwise False; read only.
Saved	VT_BOOL	Returns True if the document has never been saved, but has not changed since it was created. Returns True if it has been saved and has not changed since last saved. Returns False if it has never been saved and has changed since it was created; or if it was saved, but has changed since last saved. Read only; required.
Subject	VT_BSTR	Sets or returns summary information about the subject of the document; read/write.
Title	VT_BSTR	Sets or returns summary information about the title of the document; read/write.

The Document object should have the methods listed in the following table. The methods **Activate,**

Close, Print, Save, and **SaveAs** are required; other methods are optional.

Method name	Return type	Description
Activate	VT_EMPTY	Activates the first window associated with the document. Required.
Close	VT_EMPTY	Closes all windows associated with the document and removes the document from the Documents collection. Required. Takes two optional arguments, *saveChanges* (VT_BOOL) and *filename* (VT_BSTR). The *filename* argument specifies the name of the file in which to save the document.
NewWindow	VT_EMPTY	Creates a new window for the document.
Print	VT_EMPTY	Prints the document. Required. Takes three optional arguments: *from* (VT_I2), *to* (VT_I2), and *copies* (VT_I2). The *from* and *to* arguments specify the page range to print. The *copies* argument specifies the number of copies to print.
PrintOut	VT_EMPTY	Same as **Print** method, but provides an easier way to use the method in Visual Basic, because **Print** is a Visual Basic keyword.
PrintPreview	VT_EMPTY	Previews the pages and page breaks of the document. Equivalent to clicking **Print Preview** on the **File** menu.
RevertToSaved	VT_EMPTY	Reverts to the last saved copy of the document, and discards any changes.
Save	VT_EMPTY	Saves changes to the file specified in the document's **FullName** property. Required.
SaveAs	VT_EMPTY	Saves changes to a file. Required. Takes one optional argument, *filename* (VT_BSTR). The *filename* argument can include an optional path specification.

Documents Collection Object

If your application supports a multiple-document interface (MDI), you should provide a Documents collection object. Use the name Documents for this collection, unless the name is inappropriate for the application.

The Documents collection object should have all of the following properties.

Property name	Return type	Description
Application	VT_DISPATCH	Returns the Application object; read only. Required.
Count	VT_I4	Returns the number of items in the collection; read only. Required.
_NewEnum	VT_DISPATCH	A special property that returns an enumerator object that implements **IEnumVARIANT**. Required.
Parent	VT_DISPATCH	Returns the parent of the Documents collection object; read only. Required.

The Documents collection object should have all of the following methods.

Method name	Return type	Description
Add	VT_DISPATCH	Creates a new document and adds it to the collection. Returns the document that was created. Required.
Close	VT_EMPTY	Closes all documents in the collection. Required.
Item	VT_DISPATCH or VT_EMPTY	Returns a Document object from the collection or returns VT_EMPTY if the document does not exist. Takes an optional argument, *index*, which may be a string (VT_BSTR) indicating the document name, a number (VT_I4) indicating the ordered position within the collection, or either (VT_VARIANT). If *index* is omitted, returns the Document collection. The **Item** method is the default member (DISPID_VALUE). Required.
Open	VT_DISPATCH or VT_EMPTY	Opens an existing document and adds it to the collection. Returns the document that was opened, or VT_EMPTY if the object could not be opened. Takes one required argument, *filename*, and one optional argument, *password*. Both arguments have the type VT_BSTR. Required.

The Font Object

The Font object may be appropriate for some applications. The properties **Application, Bold, Italic, Parent,** and **Size** are required; other properties are optional. The Font object should have the following properties.

Property name	Return type	Description
Application	VT_DISPATCH	Returns the Application object; read only. Required.
Bold	VT_BOOL	Sets or returns True if the font is bold, otherwise False; read/write. Required.
Color	VT_I4	Sets or returns the RGB color of the font; read/write.
Italic	VT_BOOL	Sets or returns True if the font is italic; otherwise False, read/write. Required.
Name	VT_BSTR	Returns the name of the font; read only.
OutlineFont	VT_BOOL	Sets or returns True if the font is scaleable, otherwise False. For example, bitmapped fonts are not scaleable, whereas TrueType® fonts are scaleable; read/write.
Parent	VT_DISPATCH	Returns the parent of the Font object; read only. Required.
Shadow	VT_BOOL	Sets or returns True if the font appears with a shadow, otherwise False; read/write.
Size	VT_R4	Sets or returns the point size of the font; read/write. Required.
Strikethrough	VT_BOOL	Sets or returns True if the font appears with a line running through it, otherwise False; read/write.
Subscript	VT_BOOL	Sets or returns True if the font is subscripted, otherwise False; read/write.
Superscript	VT_BOOL	Sets or returns True if the font is superscripted, otherwise False; read/write.

Naming Conventions

Choose names for exposed objects, properties, and methods that can be easily understood by users of the application. The guidelines in this section apply to all of the following exposed items:

- Objects – implemented as classes in an application
- Properties and methods – implemented as members of a class
- Named arguments – implemented as named parameters in a member function
- Constants and enumerations – implemented as settings for properties and methods

Use entire words or syllables

It is easier for users to remember complete words than to remember whether you abbreviated Window as Wind, Wn, or Wnd.

When you need to abbreviate because an identifier would be too long, try to use complete initial syllables. For example, use AltExpEval instead of AlternateExpressionEvaluation.

Use	Don't use
Application	App
Window	Wnd

Use mixed case

All identifiers should use mixed case, rather than underscores, to separate words.

Use	Don't use
ShortcutMenus	Shortcut_Menus, Shortcutmenus, SHORTCUTMENUS, SHORTCUT_MENUS
BasedOn	basedOn

Use the same word used in the interface

Use consistent terminology. Do not use names like HWND that are based on Hungarian notation. Try to use the same word users would use to describe a concept.

Use	Don't use
Name	Lbl

Use the correct plural for the class name

Collection classes should use the correct plural for the class name. For example, if you have a class named Axis, store the collection of Axis objects in an Axes class. Similarly, a collection of Vertex objects should be stored in a Vertices class. In cases where English uses the same word for the plural, append the word "Collection."

Use	Don't use
Axes	Axiss
SeriesCollection	CollectionSeries
Windows	ColWindow

Using plurals rather than inventing new names for collections reduces the number of items a user must remember, and simplifies the selection of names for collections.

For some collections, however, this may not be appropriate, especially where a set of objects exists independently of the collection. For example, a Mail program might have a Name object that exists in several collections, such as ToList, CCList, and GroupList. In this case, you might specify the individual name collections as ToNames, CCNames, and GroupNames.

Programmability Interfaces

Embeddable objects, including ActiveX controls, often require access to the programmabilty interfaces of their containers. Similarly, containers often require access to the programmabilty interfaces of their embedded objects.

The following sections describe the standards for exposing the programmability interfaces from various components. With the advent of document objects and ActiveX controls on the Internet, adhering to these standards will become increasingly important.

Accessing the Containing Document

Objects that are embedded in a container often require access to that container's programmability interface (for example, its **IDispatch** implementation). The container should implement its document-level programmabity interface (for example, the Document object) that matches the one used by **IOleContainer** (either VTBL or **IDispatch**). To access the containing document, an object can call **IOleClientSite::GetContainer**, which returns a pointer to **IOleContainer**, and can then call **QueryInterface()** for the appropriate interface (usually **IID_IDispatch**).

Embedded objects can also access type information by using VTBL binding to dual interfaces, calling **QueryInterface** to **IOleContainer** for **IProvideClassInfo**.

Accessing the Containing Application

Embedded objects that require access to the Application object of their container (the top-level programmability object for the process) should use the **ServiceProvider** interfaces to access it.

Containers should implement **IServiceProvider** with the same implementation as with **IOleClientSite**, and at the minimum, should support **SID_Application**. If the container can also be embedded, use its container's **IServiceProvider** implementation to access the Application object. If an error occurs, embedded objects should perform a **QueryInterface** on **IOleClientSite** for **IServiceProvider** and use **IServiceProvider** to request **SID_Application**.

The standards for Automation specify that document-level objects in an application's programmabilty model should support the **Parent** and **Application** properties. If this also doesn't work (because the immediate container does not support **IServiceProivder,** or **SID_Application**), an embedded object can access the containing application by calling **IOleClientSite::GetContainer, QueryInterface(IID_IDispatch,** followed by **IDispatch::GetIDsofNames**. This gets the dispatch ID for the **Parent** or **Application** property.

Dispatch Interface and API Functions

The dispatch interfaces provide a way to expose and access objects within an application. Automation defines the following dispatch interfaces and functions.

- **IDispatch** interface − Exposes objects, methods, and properties to Automation programming tools and other applications.
- Dispatch API functions − Simplifies the implementation of the **IDispatch** interface. Use these functions to generate an **IDispatch** interface automatically.
- **IEnumVARIANT** interface − Provides a way for ActiveX clients to iterate over collection objects. This is a dispatch interface.

Overview of the IDispatch Interface

The following table describes the member functions of the **IDispatch** interface.

Interface	Member function	Purpose
IDispatch		
	GetIDsOfNames	Maps a single member name and an optional set of argument names to a corresponding set of integer dispatch IDs, which can then be used on subsequent calls to **Invoke**.
	GetTypeInfo	Retrieves the type information for an object.
	GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
	Invoke	Provides access to properties and methods exposed by an object.

Implementing the IDispatch Interface

IDispatch is located in the Oleauto.h header file on 32-bit systems, and in Dispatch.h on 16-bit systems.

ActiveX or OLE objects can implement the **IDispatch** interface for access by ActiveX clients, such as Visual Basic. The object's properties and methods can be accessed using **IDispatch::GetIDsOfNames** and **IDispatch::Invoke**.

The following examples show how to access an ActiveX or OLE object through the **IDispatch** interface. The code is abbreviated for brevity, and omits error handling.

```
// Declarations of variables used.
     DEFINE_GUID(CLSID_Hello,            // Portions omitted for brevity.

     HRESULT hresult;
     IUnknown * punk;
     IDispatch * pdisp;
     OLECHAR FAR* szMember = "SayHello";
     DISPID dispid;
     DISPPARAMS dispparamsNoArgs = {NULL, NULL, 0, 0};
     EXCEPINFO excepinfo;
     UINT nArgErr;
```

In the following code, the **OleInitialize** function loads the OLE dynamic-link libraries (DLLs), and the **CoCreateInstance** function initializes the ActiveX or OLE object's class factory. For more information on these two functions, see the *Microsoft OLE Programmer's Guide and Reference*.

```
// Initialize OLE DLLs.
hresult = OleInitialize(NULL);

// OLE function CoCreateInstance starts application using GUID.
hresult = CoCreateInstance(CLSID_Hello, NULL, CLSCTX_SERVER, IID_IUnknown,
(void FAR* FAR*)&punk);
```

QueryInterface checks whether the object supports **IDispatch**. (As with any call to **QueryInterface**, the returned pointer must be released when it is no longer needed.)

```
// Call QueryInterface to see if object supports IDispatch.
hresult = punk->QueryInterface(IID_IDispatch, &pdisp);
```

GetIDsOfNames retrieves the dispatch ID (DISPID) for the indicated method or property, in this case, szMember.

```
// Retrieve the dispatch identifier for the SayHello method.
// Use defaults where possible.
hresult = pdisp->GetIDsOfNames(
     IID_NULL,
     &szMember,
     1,
     LOCALE_USER_DEFAULT,
     &dispid);
```

In the following call to **Invoke**, the dispatch identifier (DISPID) indicates the property or method to invoke. The **SayHello** method does not take any parameters, so the fifth argument (*&dispparamsNoArgs*), contains a Null and 0, as initialized at declaration.

To invoke a property or method that requires parameters, supply the parameters in the DISPPARAMS structure.

```
// Invoke the method. Use defaults where possible.
hresult = pdisp->Invoke(
     dispid,
     IID_NULL,
     LOCALE_SYSTEM_DEFAULT,
     DISPATCH_METHOD,
     &dispparamsNoArgs,
     NULL,
     NULL,
     NULL);
```

IDispatch::GetIDsOfNames

HRESULT IDispatch::GetIDsOfNames(

REFIID *riid*,
OLECHAR FAR* FAR* *rgszNames*,
unsigned int *cNames*,
LCID *lcid*,
DISPID FAR* *rgdispid*
);

Maps a single member and an optional set of argument names to a corresponding set of integer dispatch IDs (DISPIDs), which can be used on subsequent calls to **IDispatch::Invoke**. The dispatch function **DispGetIDsOfNames** provides a standard implementation of **GetIDsOfNames**.

Parameters

riid

Reserved for future use. Must be IID_NULL.

rgszNames

Passed-in array of names to be mapped.

cNames

Count of the names to be mapped.

lcid

The locale context in which to interpret the names.

rgdispid

Caller-allocated array, each element of which contains an ID corresponding to one of the names passed in the *rgszNames* array. The first element represents the member name. The subsequent elements represent each of the member's parameters.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
DISP_E_UNKNOWNNAME	One or more of the names were not known. The returned array of dispatch IDs contains DISPID_UNKNOWN for each entry that corresponds to an unknown name.
DISP_E_UNKNOWNLCID	The locale ID (LCID) was not recognized.

Comments

An **IDispatch** implementation can associate any positive integer ID value with a given name. Zero is reserved for the default, or **Value** property; -1 is reserved to indicate an unknown name; and other

negative values are defined for other purposes. For example, if **GetIDsOfNames** is called, and the implementation does not recognize one or more of the names, it returns DISP_E_UNKNOWNNAME, and the *rgdispid* array contains DISPID_UNKNOWN for the entries that correspond to the unknown names.

The member and parameter dispatch IDs must remain constant for the lifetime of the object. This allows a client to obtain the dispatch IDs once, and cache them for later use.

When **GetIDsOfNames** is called with more than one name, the first name (*rgszNames*[0]) corresponds to the member name, and subsequent names correspond to the names of the member's parameters.

The same name may map to different dispatch IDs, depending on context. For example, a name may have a dispatch ID when it is used as a member name with a particular interface, a different ID as a member of a different interface, and different mapping for each time it appears as a parameter.

The **IDispatch** interface binds to names at run time. To bind at compile time instead, an **IDispatch** client can map names to dispatch IDs by using the type information interfaces described in Chapter 9, "Type Description Interfaces." This allows a client to bind to members at compile time and avoid calling **GetIDsOfNames** at run time. For a description of binding at compile time, see Chapter 9, "Type Description Interfaces."

The implementation of **GetIDsOfNames** is case insensitive. Users that need case-sensitive name mapping should use type information interfaces to map names to dispatch IDs (DISPIDs), rather than call **GetIDsOfNames**.

Examples

The following code from the Lines sample file Lines.cpp implements the **GetIDsOfNames** member function for the **CLine** class. The ActiveX or OLE object uses the standard implementation, **DispGetIDsOfNames**.

```
STDMETHODIMP
CLine::GetIDsOfNames(
            REFIID riid,
            OLECHAR FAR* FAR* rgszNames,
            UINT cNames,
            LCID lcid,
            DISPID FAR* rgdispid)
{
            return DispGetIDsOfNames(m_ptinfo, rgszNames, cNames, rgdispid);
}
```

The following code might appear in an ActiveX client that calls **GetIDsOfNames** to get the dispatch ID of the **CLine Color** property.

```
HRESULT hresult;
IDispatch FAR* pdisp = (IDispatch FAR*)NULL;
DISPID dispid;
OLECHAR FAR* szMember = "color";

// Code that sets a pointer to the dispatch (pdisp) is omitted.

hresult = pdisp->GetIDsOfNames(
     IID_NULL,
     &szMember,
     1, LOCALE_SYSTEM_DEFAULT,
     &dispid);
```

See Also

CreateStdDispatch, DispGetIDsOfNames, ITypeInfo::GetIDsOfNames

IDispatch::GetTypeInfo

HRESULT IDispatch::GetTypeInfo(

unsigned int *itinfo*,
LCID *lcid*,
ITypeInfo FAR* FAR* *pptinfo*
);

Retrieves the type information for an object, which can then be used to get the type information for an interface.

Parameters

itinfo

The type information to return. Pass 0 to retrieve type information for the **IDispatch** implementation.

lcid

The locale ID for the type information. An object may be able to return different type information for different languages. This is important for classes that support localized member names. For classes that do not support localized member names, this parameter can be ignored.

pptinfo

Receives a pointer to the requested type information object.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success; the type information element exists.
DISP_E_BADINDEX	Failure; *itinfo* argument was not 0.
TYPE_E_ELEMENTNOTFOUND	Failure; *itinfo* argument was not 0.

Example

The following code from the sample file Lines.cpp loads information from the type library and implements the member function **GetTypeInfo**:

```
// These lines are from CLines::Create load type information for the
// Lines collection from the type library.
     hr = LoadTypeInfo(&pLines->m_ptinfo, IID_ILines);
     if (FAILED(hr))
           goto error;

// Additional code omitted for brevity.

// This function implements GetTypeInfo for the CLines collection.
STDMETHODIMP
CLines::GetTypeInfo(
```

```
            UINT itinfo,
            LCID lcid,
            ITypeInfo FAR* FAR* pptinfo)
{
      *pptinfo = NULL;

      if(itinfo != 0)
            return ResultFromScode(DISP_E_BADINDEX);

      m_ptinfo->AddRef();
      *pptinfo = m_ptinfo;

      return NOERROR;
}
```

IDispatch::GetTypeInfoCount

HRESULT IDispatch::GetTypeInfoCount(

unsigned int FAR* *pctinfo*
);

Retrieves the number of type information interfaces that an object provides (either 0 or 1).

Parameter

pctinfo

Points to a location that receives the number of type information interfaces provided by the object. If the object provides type information, this number is 1; otherwise the number is 0.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_NOTIMPL	Failure.

Comments

The function may return zero, which indicates that the object does not provide any type information. In this case, the object may still be programmable through **IDispatch**, but does not provide type information for browsers, compilers, or other programming tools that access type information. This can be useful for hiding an object from browsers or for preventing early binding on an object.

Example

This code from the Lines sample file Lines.cpp implements the **GetTypeInfoCount** member function for the **CLines** class (ActiveX or OLE object).

```
STDMETHODIMP
CLines::GetTypeInfoCount(UINT FAR* pctinfo)
{
     *pctinfo = 1;
     return NOERROR;
}
```

See Also

CreateStdDispatch

IDispatch::Invoke

HRESULT IDispatch::Invoke(

DISPID *dispidMember*,
REFIID *riid*,
LCID *lcid*,
WORD *wFlags*,
DISPPARAMS FAR* *pdispparams*,
VARIANT FAR* *pvarResult*,
EXCEPINFO FAR* *pexcepinfo*,
unsigned int FAR* *puArgErr*
);

Provides access to properties and methods exposed by an object. The dispatch function **DispInvoke** provides a standard implementation of **IDispatch::Invoke**.

Parameters

dispidMember

Identifies the member. Use **GetIDsOfNames** or the object's documentation to obtain the dispatch identifier.

riid

Reserved for future use. Must be IID_NULL.

lcid

The locale context in which to interpret arguments. The *lcid* is used by the **GetIDsOfNames** function, and is also passed to **Invoke** to allow the object to interpret its arguments specific to a locale.

Applications that do not support multiple national languages can ignore this parameter. For more information, refer to "Supporting Multiple National Languages" in Chapter 2, "Exposing ActiveX objects."

wFlags

Flags describing the context of the **Invoke** call, as follows:

Value	Description
DISPATCH_METHOD	The member is invoked as a method. If a property has the same name, both this and the DISPATCH_PROPERTYGET flag may be set.
DISPATCH_PROPERTYGET	The member is retrieved as a property or data member.
DISPATCH_PROPERTYPUT	The member is changed as a property or data member.
DISPATCH_PROPERTYPUTREF	The member is changed by a reference assignment, rather than a value assignment. This flag is valid only when the property accepts a reference to an object.

pdispparams

Pointer to a structure containing an array of arguments, an array of argument dispatch IDs for named arguments, and counts for the number of elements in the arrays. See the Comments section that follows for a description of the DISPPARAMS structure.

pvarResult

Pointer to the location where the result is to be stored, or Null if the caller expects no result. This argument is ignored if DISPATCH_PROPERTYPUT or DISPATCH_PROPERTYPUTREF is specified.

pexcepinfo

Pointer to a structure that contains exception information. This structure should be filled in if DISP_E_EXCEPTION is returned. Can be Null.

puArgErr

The index within *rgvarg* of the first argument that has an error. Arguments are stored in *pdispparams->rgvarg* in reverse order, so the first argument is the one with the highest index in the array. This parameter is returned only when the resulting return value is DISP_E_TYPEMISMATCH or DISP_E_PARAMNOTFOUND. For details, see "Returning Errors" in the following Comments section.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
DISP_E_BADPARAMCOUNT	The number of elements provided to DISPPARAMS is different from the number of arguments accepted by the method or property.
DISP_E_BADVARTYPE	One of the arguments in *rgvarg* is not a valid variant type.
DISP_E_EXCEPTION	The application needs to raise an exception. In this case, the structure passed in *pexcepinfo* should be filled in.
DISP_E_MEMBERNOTFOUND	The requested member does not exist, or the call to **Invoke** tried to set the value of a read-only property.
DISP_E_NONAMEDARGS	This implementation of **IDispatch** does not support named arguments.
DISP_E_OVERFLOW	One of the arguments in *rgvarg* could not be coerced to the specified type.
DISP_E_PARAMNOTFOUND	One of the parameter dispatch IDs does not correspond to a parameter on the method. In this case, *puArgErr* should be set to the first argument that contains the error.
DISP_E_TYPEMISMATCH	One or more of the arguments could not be coerced. The index within *rgvarg* of the first parameter with the incorrect type is returned in the *puArgErr* parameter.
DISP_E_UNKNOWNINTERFACE	The interface ID passed in *riid* is not IID_NULL.

DISP_E_UNKNOWNLCID	The member being invoked interprets string arguments according to the locale ID (LCID), and the LCID is not recognized. If the LCID is not needed to interpret arguments, this error should not be returned.
DISP_E_PARAMNOTOPTIONAL	A required parameter was omitted.

In 16-bit versions, you can define your own errors using the MAKE_SCODE value macro.

Comments

Generally, you should not implement **Invoke** directly. Instead, use the dispatch interface create functions **CreateStdDispatch** and **DispInvoke**. For details, refer to "CreateStdDispatch" and "DispInvoke" in this chapter, and "**Creating** the IDispatch Interface" in Chapter 2, "Exposing ActiveX objects."

If some application-specific processing needs to be performed before calling a member, the code should perform the necessary actions, and then call **ITypeInfo::Invoke** to invoke the member. **ITypeInfo::Invoke** acts exactly like **IDispatch::Invoke**. The standard implementations of **IDispatch::Invoke** created by **CreateStdDispatch** and **DispInvoke** defer to **ITypeInfo::Invoke**.

In an ActiveX client, **IDispatch::Invoke** should be used to get and set the values of properties, or to call a method of an ActiveX object. The *dispidMember* argument identifies the member to invoke. The dispatch IDs that identify members are defined by the implementor of the object and can be determined by using the object's documentation, the **IDispatch::GetIDsOfNames** function, or the **ITypeInfo** interface.

The information that follows addresses developers of ActiveX clients and others who use code to expose ActiveX objects. It describes the behavior that users of exposed objects should expect.

Calling a Method With No Arguments

The simplest use of **Invoke** is to call a method that does not have any arguments. You only need to pass the dispatch ID of the method, a locale ID, the DISPATCH_METHOD flag, and an empty DISPPARAMS structure. For example:

```
HRESULT hresult;
IUnknown FAR* punk;
IDispatch FAR* pdisp = (IDispatch FAR*)NULL;
OLECHAR FAR* szMember = "Simple";
DISPID dispid;
DISPPARAMS dispparamsNoArgs = {NULL, NULL, 0, 0};

hresult = CoCreateInstance(CLSID_CMyObject, NULL, CLSCTX_SERVER,
                      IID_Unknown, (void FAR* FAR*)&punk);

hresult = punk->QueryInterface(IID_IDispatch,
                      (void FAR* FAR*)&pdisp);

hresult = pdisp->GetIDsOfNames(IID_NULL, &szMember, 1,
                      LOCALE_USER_DEFAULT, &dispid);

hresult = pdisp->Invoke(
           dispid,
           IID_NULL,
           LOCALE_USER_DEFAULT,
```

```
            DISPATCH_METHOD,
            &dispparamsNoArgs, NULL, NULL, NULL);
```

The example invokes a method named **Simple** on an object of the class **CMyObject**. First, it calls **CoCreateInstance**, which instantiates the object and returns a pointer to the object's **IUnknown** interface (punk). Next, it calls **QueryInterface**, receiving a pointer to the object's **IDispatch** interface (pdisp). It then uses pdisp to call the object's **GetIDsOfNames** function, passing the string **Simple** in szMember to get the dispatch ID for the **Simple** method. With the dispatch ID for **Simple** in dispid, it calls **Invoke** to invoke the method, specifying DISPATCH_METHOD for the wFlags parameter and using the system default locale.

To further simplify the code, the example declares a DISPPARAMS structure named dispparamsNoArgs that is appropriate to an **Invoke** call with no arguments.

Because the **Simple** method does not take any arguments and does not return a result, the puArgErr and pvarResult parameters are Null. In addition, the example passes Null for pexcepinfo, indicating that it is not prepared to handle exceptions and will handle only HRESULT errors.

Most methods, however, take one or more arguments. To invoke these methods, the DISPPARAMS structure should be filled in, as described in "Passing Parameters" later in this chapter.

Automation defines special dispatch IDs for invoking an object's **Value** property (the default), and the memebers **_NewEnum,** and **Evaluate**. For details, see "DISPID" in Chapter 6, "Data Types, Structures, and Enumerations."

Getting and Setting Properties

Properties are accessed in the same way as methods, except you specify DISPATCH_PROPERTYGET or DISPATCH_PROPERTYPUT instead of DISPATCH_METHOD. Some languages can not distinguish between retrieving a property and calling a method. In this case, you should set the flags DISPATCH_PROPERTYGET and DISPATCH_METHOD.

The following example gets the value of a property named **On.** You can assume that the object has been created, and that its interfaces have been queried, as in the previous example.

```
VARIANT FAR *pvarResult;
// Code omitted for brevity.
szMember = "On";
hresult = pdisp->GetIDsOfNames(IID_NULL, &szMember, 1,
                      LOCALE_USER_DEFAULT, &dispid);

hresult = pdisp->Invoke(
           dispid,
           IID_NULL,
           LOCALE_USER_DEFAULT,
           DISPATCH_PROPERTYGET,
           &dispparamsNoArgs, pvarResult, NULL, NULL);
```

As in the previous example, the code calls **GetIDsOfNames** for the dispatch ID of the **On** property, and then passes the ID to **Invoke**. Then, **Invoke** returns the property's value in **pvarResult**. In general, the return value does not set VT_BYREF. However, this bit may be set and a pointer returned to the return value, if the lifetime of the return value is the same as that of the object.

To change the property's value, the call looks like this:

```
VARIANT FAR *pvarResult;
DISPPARAMS dispparams;
DISPID mydispid = DISP_PROPERTYPUT
```

```
// Code omitted for brevity.

szMember = "On";
dispparams.rgvarg[0].vt = VT_BOOL;
dispparams.rgvarg[0].bool = FALSE;
dispparams.rgdispidNamedArgs = &mydispid;
dispparams.cArgs = 1;
dispparams.cNamedArgs = 1;
hresult = pdisp->GetIDsOfNames(IID_NULL, &szMember, 1,
                       LOCALE_USER_DEFAULT, &dispid);

hresult = pdisp->Invoke(
           dispid,
           IID_NULL,
           LOCALE_USER_DEFAULT,
           DISPATCH_PROPERTYPUT,
           &dispparams, NULL, NULL, NULL);
```

The new value for the property (the Boolean value False) is passed as an argument when the **On** property's **Put** function is invoked. The dispatch ID for the argument is DISPID_PROPERTYPUT. This dispatch ID is defined by Automation to designate the parameter that contains the new value for a property's **Put** function. The remaining details of the DISPPARAMS structure are described in the next section, "Passing Parameters."

The DISPATCH_PROPERTYPUT flag in the previous example indicates that a property is being set by value. In Visual Basic, the following statement assigns the **Value** property (the default)of **YourObj** to the **Prop** property:

```
MyObj.Prop = YourObj
```

This statement should be flagged as a DISPATCH_PROPERTYPUT. Similarly, statements like the following assign the **Value** property of one object to the **Value** property of another object.

```
Worksheet.Cell(1,1) = Worksheet.Cell(6,6)
MyDoc.Text1 = YourDoc.Text1
```

These statements result in a PROPERTY_PUT operation on `Worksheet.Cell(1,1)` and `MyDoc.Text1`.

Use the DISPATCH_PROPERTYPUTREF flag to indicate a property or data member that should be set by reference. For example, the following Visual Basic statement assigns the pointer **YourObj** to the property **Prop,** and should be flagged as DISPATCH_PROPERTYPUTREF.

```
Set MyObj.Prop = YourObj
```

The **Set** statement causes a reference assignment, rather than a value assignment.

The parameter on the right side is always passed by name, and should not be accessed positionally.

Passing Parameters

Arguments to the method or property being invoked are passed in the DISPPARAMS structure. This structure consists of a pointer to an array of arguments represented as variants, a pointer to an array of dispatch IDs for named arguments, and the number of arguments in each array.

```
typedef struct FARSTRUCT tagDISPPARAMS{
     VARIANTARG FAR* rgvarg;             // Array of arguments.
```

```
    DISPID FAR* rgdispidNamedArgs;      // Dispatch IDs of named arguments.
    unsigned int cArgs;                       // Number of arguments.
    unsigned int cNamedArgs;            // Number of named arguments.
} DISPPARAMS;
```

The arguments are passed in the array *rgvarg*[ ], with the number of arguments passed in *cArgs*. The arguments in the array should be placed from last to first, so *rgvarg*[0] has the last argument and *rgvarg*[*cArgs* -1] has the first argument. The method or property may change the values of elements within the array *rgvarg*, but only if it has set the VT_BYREF flag. Otherwise, consider the elements as read-only.

A dispatch invocation can have named arguments as well as positional arguments. If *cNamedArgs* is 0, all the elements of *rgvarg*[ ] represent positional arguments. If *cNamedArgs* is not 0, each element of *rgdispidNamedArgs*[ ] contains the dispatch ID of a named argument, and the value of the argument is in the matching element of *rgvarg*[ ]. The dispatch IDs of the named arguments are always contiguous in *rgdispidNamedArgs*, and their values are in the first *cNamedArgs* elements of *rgvarg*. Named arguments cannot be accessed positionally, and positional arguments cannot be named.

The dispatch ID of an argument is its zero-based position in the argument list. For example, the following method takes three arguments.

```
BOOL _export CDECL
CCredit::CheckCredit(BSTR bstrCustomerID,// DISPID = 0.
                             BSTR bstrLenderID,          // DISPID = 1.
                             CURRENCY cLoanAmt)          // DISPID = 2.
{
// Code omitted.
}
```

If you include the dispatch ID with each named argument, you can pass the named arguments to **Invoke** in any order. For example, if a method is to be invoked with two positional arguments, followed by three named arguments (*A, B,* and *C*), using the following hypothetical syntax, then *cArgs* would be 5, and *cNamedArgs* would be 3.

```
object.method("arg1", "arg2", A := "argA", B := "argB", C := "argC")
```

The first positional argument would be in *rgvarg*[4]. The second positional argument would be in *rgvarg*[3]. The ordering of named arguments is not important to the **IDispatch** implementation, but these arguments are generally passed in reverse order. The argument *A* would be in *rgvarg*[2], with the dispatch ID of *A* in *rgdispidNamedArgs*[2]. The argument *B* would be in *rgvarg*[1], with the corresponding dispatch ID in *rgdispidNamedArgs*[1]. The argument *C* would be in *rgvarg*[0], with the dispatch ID corresponding to *C* in *rgdispidNamedArgs*[0]. The following diagram illustrates the arrays and their contents.

{ewc msdncd, EWGraphic, bsd23527 0 /a "SDK_01.WMF"}

You can also use **Invoke** on members with optional arguments, but all optional arguments must be of type VARIANT. As with required arguments, the contents of the argument vector depend on whether the arguments are positional or named. The invoked member must ensure that the arguments are valid. **Invoke** merely passes the DISPPARAMS structure it receives.

Omitting named arguments is straightforward. You would pass the arguments in *rgvarg* and their dispatch IDs in *rgdispidNamedArgs*. To omit the argument named *B* (in the preceding example) you would set *rgvarg*[0] to the value of *C*, with its dispatch ID in *rgdispidNamedArgs*[0]; and *rgvarg*[1] to the value of *A,* with its dispatch ID in *rgdispidNamedArgs*[1]. The subsequent positional arguments would occupy elements 2 and 3 of the arrays. In this case, *cArgs* is 4 and *cNamedArgs* is 2.

If the arguments are positional (unnamed), you would set *cArgs* to the total number of possible arguments, *cNamedArgs* to 0, and pass VT_ERROR as the type of the omitted arguments, with the status code DISP_E_PARAMNOTFOUND as the value. For example, the following code invokes `ShowMe(,1)`.

```
VARIANT FAR *pvarResult;
EXCEPINFO FAR *pexcepinfo;
unsigned int FAR *puArgErr;
DISPPARAMS dispparams;

// Code omitted for brevity.

szMember = "ShowMe";
hresult = pdisp->GetIDsOfNames(IID_NULL, &szMember, 1,
                                                LOCALE_USER_DEFAULT, &dispid)
;
dispparams.rgvarg[0].vt = VT_I2;
dispparams.rgvarg[0].ival = 1;
dispparams.rgvarg[1].vt = VT_ERROR;
dispparams.rgvarg[1].scode = DISP_E_PARAMNOTFOUND;
dispparams.cArgs = 2;
dispparams.cNamedArgs = 0;

hresult = pdisp->Invoke(
           dispid,
           IID_NULL,
           LOCALE_USER_DEFAULT,
           DISPATCH_METHOD,
           &dispparams, pvarResult, pexcepinfo, puArgErr);
```

The example takes two positional arguments, but omits the first. Therefore, *rgvarg*[0] contains 1, the value of the last argument in the argument list, and *rgvarg*[1] contains VT_ERROR and the error return value, indicating the omitted first argument.

The calling code is responsible for releasing all strings and objects referred to by *rgvarg*[ ] or placed in **pvarResult*. As with other parameters that are passed by value, if the invoked member must maintain access to a string after returning, you should copy the string. Similarly, if the member needs access to a passed-object pointer after returning, it must call the **AddRef** function on the object. A common example occurs when an object property is changed to refer to a new object, using the DISPATCH_PROPERTYPUTREF flag.

For those implementing **IDispatch::Invoke,** Automation provides the **DispGetParam** function to retrieve parameters from the argument vector and coerce them to the proper type. For details, see "DispGetParam" later in this chapter.

Indexed Properties

When you invoke indexed properties of any dimension, you must pass the indexes as additional arguments. To set an indexed property, place the new value in the first element of the *rgvarg*[ ] vector, and the indexes in the subsequent elements. To get an indexed property, pass the indexes in the first *n* elements of *rgvarg*, and the number of indexes in *cArg*. **Invoke** returns the value of the property in *pvarResult*.

Automation stores array data in column-major order, which is the same ordering scheme used by Visual Basic and FORTRAN, but different from C, C++, and Pascal. If you are programming in C, C++, or Pascal, you must pass the indexes in the reverse order. The following example shows how to fill the DISPPARAMS structure in C++.

```
dispparams.rgvarg[0].vt = VT_I2;
dispparams.rgvarg[0].iVal = 99;
dispparams.rgvarg[1].vt = VT_I2;
dispparams.rgvarg[1].iVal = 2;
dispparams.rgvarg[2].vt = VT_I2;
dispparams.rgvarg[2].iVal = 1;
dispparams.rgdispidNamedArgs = DISPID_PROPERTYPUT;
dispparams.cArgs = 3;
dispparams.cNamedArgs = 1;
```

The example changes the value of Prop[1,2] to 99. The new property value is passed in *rgvarg*[0]. The right-most index is passed in `rgvarg[1]`, and the next index in `rgvarg[2]`. The `cArgs` field specifies the number of elements of `rgvarg[ ]` that contain data, and `cNamedArgs` is `1`, indicating the new value for the property.

Property collections are an extension of this feature.

Raising Exceptions During Invoke

When you implement **IDispatch::Invoke,** errors can be communicated either through the normal return value or by raising an exception. An exception is a special situation that is normally handled by jumping to the nearest routine enclosing the exception handler.

To raise an exception, **IDispatch::Invoke** returns DISP_E_EXCEPTION and fills the structure passed through *pexcepinfo* with information about the cause of the exception or error. You can use the information to understand the cause of the exception and proceed as necessary.

The exception information structure includes an error code number that identifies the kind of exception (a string that describes the error in a human-readable way). It also includes a Help file and a Help context number that can be passed to Windows Help for details about the error. At a minimum, the error code number must be filled with a valid number.

If you consider **IDispatch** another way to call C++ methods in an interface, EXCEPINFO models the raising of an exception or **longjmp()** call by such a method.

Returning Errors

Invoke returns DISP_E_MEMBERNOTFOUND if one of the following conditions occurs:

- A member or parameter with the specified dispatch ID and matching *cArgs* cannot be found, and the parameter is not optional.
- The member is a void function, and the caller did not set *pvarResult* to Null.
- The member is a read-only property, and the caller set *wFlags* to DISPATCH_PROPERTYPUT or DISPATCH_PROPERTYPUTREF.

If **Invoke** finds the member, but uncovers errors in the argument list, it returns one of several other errors. DISP_E_BAD_PARAMCOUNT means that the DISPPARAMS structure contains an incorrect number of parameters for the property or method. DISP_E_NONAMEDARGS means that **Invoke** received named arguments, but they are not supported by the member.

DISP_E_PARAMNOTFOUND means that the correct number of parameters was passed, but the dispatch ID for one or more parameters was incorrect. If **Invoke** cannot convert one of the arguments to the desired type, it returns DISP_E_TYPEMISMATCH. In these two cases, if it can identify which argument is incorrect, **Invoke** sets **puArgErr* to the index within *rgvarg* of the argument with the error. For example, if an Automation method expects a reference to a double-precision number as an argument, but receives a reference to an integer, the argument is coerced. However, if the method receives a date,

IDispatch::Invoke returns DISP_E_TYPEMISMATCH and sets **puArgErr* to the index of the integer in the argument array.

Automation provides functions to perform standard conversions of VARIANT, and these functions should be used for consistent operation. DISP_E_TYPEMISMATCH is returned only when these functions fail. For more information about converting arguments, see Chapter 7, "Conversion and Manipulation Functions."

Example

This code from the Lines sample file Lines.cpp implements the **Invoke** member function for the **CLines** class.

```
STDMETHODIMP
CLines::Invoke(
      DISPID dispidMember,
      REFIID riid,
      LCID lcid,
      WORD wFlags,
      DISPPARAMS FAR* pdispparams,
      VARIANT FAR* pvarResult,
      EXCEPINFO FAR* pexcepinfo,
      UINT FAR* puArgErr)
{
      return DispInvoke(
            this, m_ptinfo,
            dispidMember, wFlags, pdispparams,
            pvarResult, pexcepinfo, puArgErr);
}
```

The next code example calls the **CLines::Invoke** member function to get the value of the **Color** property:

```
HRESULT hr;
EXCEPINFO excepinfo;
UINT nArgErr;
VARIANT vRet;
DISPPARAMS FAR* pdisp;
OLECHAR FAR* szMember;
DISPPARAMS dispparamsNoArgs = {NULL, NULL, 0, 0};

// Initialization code omitted for brevity.
szMember = "Color";
hr = pdisp->GetIDsOfNames(IID_NULL, &szMember, 1, LOCALE_USER_DEFAULT,
      &dispid);

// Get Color property.
hr = pdisp->Invoke(dispid, IID_NULL, LOCALE_SYSTEM_DEFAULT,
      DISPATCH_PROPERTYGET, &dispparams, &vRet, &excepinfo, &nArgErr);
```

See Also

CreateStdDispatch, **DispInvoke**, **DispGetParam**, **ITypeInfo::Invoke**

Overview of Dispatch API Functions

For 32-bit systems, dispatch functions are provided in the file Oleaut32.dll. The header file is Oleauto.h, and the import library is Oleaut32.lib. For 16-bit systems, the dispatch functions are provided in the file Oledisp.dll. The header file is Dispatch.h, and the import library is Ole2disp.lib. These functions simplify the creation of **IDispatch** interfaces. The dispatch functions are summarized in the following table.

Category	Function name	Purpose
Dispatch interface creation	**CreateDispTypeInfo**	Creates simplified type information for an object.
	CreateStdDispatch	Creates a standard **IDispatch** implementation for an object.
	DispGetIDsOfNames	Converts a set of names to dispatch IDs.
	DispGetParam	Retrieves and coerces elements from a DISPPARAMS structure.
	DispInvoke	Calls a member function of an **IDispatch** interface.
Active object initialization	**GetActiveObject**	Retrieves an instance of an object that is initialized with OLE.
	RegisterActiveObject	Initializes a running object with OLE. (Use when application starts.)
	RevokeActiveObject	Revokes a running application's initialization with OLE. (Use when application ends.)

Using API Functions with the IDispatch Interface

The following five functions are used to create and modify **IDispatch**.

CreateDispTypeInfo Quick Info

HRESULT CreateDispTypeInfo (

INTERFACEDATA *pInterfacedata*,
LCID *lcid*,
ITypeInfo FAR* FAR* *pptinfo*
);

Creates simplified type information for use in an implementation of **IDispatch**.

Parameters

pInterfacedata

The interface description that this type information describes.

lcid

The locale ID (LCID) for the names used in the type information.

pptinfo

On return, pointer to a type information implementation for use in **DispGetIDsOfNames** and **DispInvoke**.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	The interface is supported.
E_INVALIDARG	Either the interface description or the locale ID is invalid.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

Comments

You can construct type information at run time by using **CreateDispTypeInfo** and an INTERFACEDATA structure that describes the object being exposed.

The type information returned by this function is primarily designed to automate the implementation of **IDispatch**. **CreateDispTypeInfo** does not return all of the type information described in Chapter 9, "Type Description Interfaces." The argument *pInterfaceData* is not a complete description of an interface. It does not include Help information, comments, optional parameters, and other type information that is useful in different contexts.

Accordingly, the recommended method for providing type information about an object is to describe the object using the Object Description Language (ODL), and to compile the object description into a type library using the MIDL compiler or the MkTypLib utility.

To use type information from a type library, use the **LoadTypeLib** and **GetTypeInfoOfGuid** functions instead of **CreateDispTypeInfo**. For more information, see Chapter 9, "Type Description Interfaces."

Example

The code that follows creates type information from INTERFACEDATA to expose the **CCalc** object.

```
static METHODDATA NEARDATA rgmdataCCalc[] =
{
            PROPERTY(VALUE,  IMETH_ACCUM,    IDMEMBER_ACCUM,    VT_I4)
            PROPERTY(ACCUM,  IMETH_ACCUM,    IDMEMBER_ACCUM,    VT_I4)
            PROPERTY(OPND,   IMETH_OPERAND,  IDMEMBER_OPERAND,  VT_I4)
            PROPERTY(OP,     IMETH_OPERATOR, IDMEMBER_OPERATOR, VT_I2)
            METHOD0(EVAL,    IMETH_EVAL,     IDMEMBER_EVAL,     VT_BOOL)
            METHOD0(CLEAR,   IMETH_CLEAR,    IDMEMBER_CLEAR,    VT_EMPTY)
            METHOD0(DISPLAY, IMETH_DISPLAY,  IDMEMBER_DISPLAY,  VT_EMPTY)
            METHOD0(QUIT,    IMETH_QUIT,     IDMEMBER_QUIT,     VT_EMPTY)
            METHOD1(BUTTON,  IMETH_BUTTON,   IDMEMBER_BUTTON,   VT_BOOL)
};

INTERFACEDATA NEARDATA g_idataCCalc =
{
     rgmdataCCalc, DIM(rgmdataCCalc)
};

// Use Dispatch interface API functions to implement IDispatch.
CCalc FAR*
CCalc::Create()
{
     HRESULT hresult;
     CCalc FAR* pcalc;
     CArith FAR* parith;
     ITypeInfo FAR* ptinfo;
     IUnknown FAR* punkStdDisp;
extern INTERFACEDATA NEARDATA g_idataCCalc;

     if((pcalc = new FAR CCalc()) == NULL)
           return NULL;
     pcalc->AddRef();

     parith = &(pcalc->m_arith);

     // Build type information for the functionality on this object that
     // is being exposed for external programmability.
     hresult = CreateDispTypeInfo(
           &g_idataCCalc, LOCALE_SYSTEM_DEFAULT, &ptinfo);
     if(hresult != NOERROR)
           goto LError0;

     // Create an aggregate with an instance of the default
     // implementation of IDispatch that is initialized with
     // type information.
     hresult = CreateStdDispatch(
           pcalc,                        // Controlling unknown.
           parith,                       // Instance to dispatch on.
           ptinfo,                       // Type information describing the
instance.
           &punkStdDisp);
```

```
     ptinfo->Release();

     if(hresult != NOERROR)
           goto LError0;

     pcalc->m_punkStdDisp = punkStdDisp;

     return pcalc;

LError0:;
     pcalc->Release();
     return NULL;
}
```

CreateStdDispatch Quick Info

HRESULT CreateStdDispatch (

IUnknown FAR* *punkOuter*,
void FAR* *pvThis*,
ITypeInfo FAR* *ptinfo*,
IUnknown FAR* FAR* *ppunkStdDisp*
);

Creates a standard implementation of the **IDispatch** interface through a single function call. This simplifies exposing objects through Automation.

Parameters

punkOuter

Pointer to the object's **IUnknown** implementation.

pvThis

Pointer to the object to expose.

ptinfo

Pointer to the type information that describes the exposed object.

ppunkStdDisp

This is the private unknown for the object that implements the **IDispatch** interface **QueryInterface** call. This pointer is Null if the function fails.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_INVALIDARG	One of the first three arguments is invalid.
E_OUTOFMEMORY	There was insufficient memory to complete the operation.

Comments

You can use **CreateStdDispatch** when creating an object instead of implementing the **IDispatch** member functions for the object. However, the implementation that **CreateStdDispatch** creates has these limitations:

- Supports only one national language.
- Supports only dispatch-defined exception codes returned from **Invoke**.

LoadTypeLib, **GetTypeInfoOfGuid**, and **CreateStdDispatch** comprise the minimum set of functions that you need to call to expose an object using a type library. For more information on **LoadTypeLib** and **GetTypeInfoOfGuid**, see Chapter 9, "Type Description Interfaces."

CreateDispTypeInfo and **CreateStdDispatch** comprise the minimum set of dispatch components you

need to call to expose an object using type information provided by the INTERFACEDATA structure.

Example

The following code implements the **IDispatch** interface for the **CCalc** class using **CreateStdDispatch**.

```
CCalc FAR*
CCalc::Create()
{
     HRESULT hresult;
     CCalc FAR* pcalc;
     CArith FAR* parith;
     ITypeInfo FAR* ptinfo;
     IUnknown FAR* punkStdDisp;
extern INTERFACEDATA NEARDATA g_idataCCalc;

     if((pcalc = new FAR CCalc()) == NULL)
          return NULL;
     pcalc->AddRef();

     parith = &(pcalc->m_arith);

     // Build type information for the functionality on this object that
     // is being exposed for external programmability.
     hresult = CreateDispTypeInfo(
          &g_idataCCalc, LOCALE_SYSTEM_DEFAULT, &ptinfo);
     if(hresult != NOERROR)
          goto LError0;

     // Create an aggregate with an instance of the default
     // implementation of IDispatch that is initialized with
     // type information.
     hresult = CreateStdDispatch(
          pcalc,                         // Controlling unknown.
          parith,                        // Instance to dispatch on.
          ptinfo,                        // Type information describing the
instance.
     &punkStdDisp);

     ptinfo->Release();

     if(hresult != NOERROR)
          goto LError0;

     pcalc->m_punkStdDisp = punkStdDisp;

     return pcalc;

LError0:;
     pcalc->Release();
     return NULL;
}
```

DispGetIDsOfNames Quick Info

HRESULT DispGetIDsOfNames (

ITypeInfo* *ptinfo*,
OLECHAR FAR* FAR* *rgszNames*,
unsigned int *cNames*,
DISPID FAR* *rgdispid*
);

Uses type information to convert a set of names to dispatch IDs. This is the recommended implementation of **IDispatch::GetIDsOfNames**.

Parameters

ptinfo

Pointer to the type information for an interface. This type information is specific to one interface and language code, so it is not necessary to pass an interface identifier (IID) or locale class identifier (LCID) to this function.

rgszNames

An array of name strings that can be the same array passed to **DispInvoke** in the DISPPARAMS structure. If *cNames* is greater than 1, the first name is interpreted as a method name, and subsequent names are interpreted as parameters to that method.

cNames

The number of elements in *rgszNames*.

rgdispid

Pointer to an array of dispatch IDs to be filled in by this function. The first ID corresponds to the method name. Subsequent IDs are interpreted as parameters to the method.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	The interface is supported.
E_INVALIDARG	One of the arguments is invalid.
DISP_E_UNKNOWNNAME	One or more of the given names were not known. The returned array of dispatch IDs contains DISPID_UNKNOWN for each entry that corresponds to an unknown name.
Other return codes	Any of the **ITypeInfo::Invoke** errors can also be returned.

Example

This code from the Lines sample file Points.cpp implements the member function **GetIDsOfNames** for the **CPoints** class using **DispGetIDsOfNames**.

```
STDMETHODIMP
CPoints::GetIDsOfNames(
           REFIID riid,
           char FAR* FAR* rgszNames,
           UINT cNames,
           LCID lcid,
           DISPID FAR* rgdispid)
{
     return DispGetIDsOfNames(m_ptinfo, rgszNames, cNames, rgdispid);
}
```

See Also

CreateStdDispatch, **IDispatch::GetIDsOfNames**

DispGetParam Quick Info

HRESULT DispGetParam(

DISPPARAMS FAR* *dispparams*,
unsigned int *iPosition*,
VARTYPE *vt*,
VARIANT FAR* *pvarResult*,
unsigned int FAR* *puArgErr*
);

Retrieves a parameter from the DISPPARAMS structure, checking both named parameters and positional parameters, and coerces the parameter to the specified type.

Parameters

dispparams

Pointer to the parameters passed to **IDispatch::Invoke**.

iPosition

The position of the parameter in the parameter list. **DispGetParam** starts at the end of the array, so if *iPosition* is 0, the last parameter in the array is returned.

vt

The type the argument should be coerced to.

pvarResult

Pointer to the variant to pass the parameter into.

puArgErr

On return, pointer to the index of the argument that caused a DISP_E_TYPEMISMATCH error. This pointer is returned to **Invoke** to indicate the position of the argument in DISPPARAMS that caused the error.

Return Value

The return value obtained from the HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
DISP_E_BADVARTYPE	The variant type *vt* is not supported.
DISP_E_OVERFLOW	The retrieved parameter could not be coerced to the specified type.
DISP_E_PARAMNOTFOUND	The parameter indicated by *iPosition* could not be found.
DISP_E_TYPEMISMATCH	The argument could not be coerced to the specified type.
E_INVALIDARG	One of the arguments was invalid.
E_OUTOFMEMORY	Insufficient memory to complete operation.

Comments

The output parameter *pvarResult* must be a valid variant. Any existing contents are released in the standard way. The contents of the variant are freed with **VariantFree**.

If you have used **DispGetParam** to get the right side of a property put operation, the second parameter should be DISPID_PROPERTYPUT. For example:

```
DispGetParam(&dispparams, DISPID_PROPERTYPUT, VT_BOOL, &varResult)
```

Named parameters cannot be accessed positionally, and vice versa.

Example

The following example uses **DispGetParam** to set X and Y properties:

```
STDMETHODIMP
CPoint::Invoke(
      DISPID dispidMember,
      REFIID riid,
      LCID lcid,
      unsigned short wFlags,
      DISPPARAMS FAR* pdispparams,
      VARIANT FAR* pvarResult,
      EXCEPINFO FAR* pexcepinfo,
      unsigned int FAR* puArgErr)
{
      unsigned int uArgErr;
      HRESULT hresult;
      VARIANTARG varg0;
      VARIANT varResultDummy;

      UNUSED(lcid);
      UNUSED(pexcepinfo);

      // Make sure the wFlags are valid.
      if(wFlags & ~(DISPATCH_METHOD | DISPATCH_PROPERTYGET |
            DISPATCH_PROPERTYPUT | DISPATCH_PROPERTYPUTREF))
            return ResultFromScode(E_INVALIDARG);

      // This object only exposes a "default" interface.
      if(!IsEqualIID(riid, IID_NULL))
            return ResultFromScode(DISP_E_UNKNOWNINTERFACE);

      // It simplifies the following code if the caller
      // ignores the return value.
      if(puArgErr == NULL)
            puArgErr = &uArgErr;
      if(pvarResult == NULL)
            pvarResult = &varResultDummy;

      VariantInit(&varg0);

      // Assume the return type is void, unless otherwise is found.
      VariantInit(pvarResult);

      switch(dispidMember){
```

```
	case IDMEMBER_CPOINT_GETX:
		V_VT(pvarResult) = VT_I2;
		V_I2(pvarResult) = GetX();
		break;

	case IDMEMBER_CPOINT_SETX:
		hresult = DispGetParam(pdispparams, 0, VT_I2, &varg0, puArgErr);
		if(hresult != NOERROR)
			return hresult;
		SetX(V_I2(&varg0));
		break;

	case IDMEMBER_CPOINT_GETY:
		V_VT(pvarResult) = VT_I2;
		V_I2(pvarResult) = GetY();
		break;

	case IDMEMBER_CPOINT_SETY:
		hresult = DispGetParam(pdispparams, 0, VT_I2, &varg0, puArgErr);
		if(hresult != NOERROR)
			return hresult;
		SetY(V_I2(&varg0));
		break;

	default:
		return ResultFromScode(DISP_E_MEMBERNOTFOUND);
	}
	return NOERROR;
}
```

See Also

CreateStdDispatch, IDispatch::Invoke

DispInvoke Quick Info

HRESULT DispInvoke(

void FAR* *_this*,
ITypeInfo FAR* *ptinfo*,
DISPID *dispidMember*,
unsigned short *wFlags*,
DISPPARAMS FAR* *pparams*,
VARIANT FAR* *pvarResult*,
EXCEPINFO *pexcepinfo*,
unsigned int FAR* *puArgErr*
);

Automatically calls member functions on an interface, given the type information for the interface. You can describe an interface with type information and implement **IDispatch::Invoke** for the interface using this single call.

Parameters

_this

Pointer to an implementation of the **IDispatch** interface described by *ptinfo*.

ptinfo

Pointer to the type information that describes the interface.

dispidMember

Identifies the member. Use **GetIDsOfNames** or the object's documentation to obtain the dispatch ID.

wFlags

Flags describing the context of the **Invoke** call, as follows:

Value	Description
DISPATCH_METHOD	The member is invoked as a method. If a property has the same name, both this and the DISPATCH_PROPERTYGET flag can be set.
DISPATCH_PROPERTYGET	The member is retrieved as a property or data member.
DISPATCH_PROPERTYPUT	The member is changed as a property or data member.
DISPATCH_PROPERTYPUTREF	The member is changed by a reference assignment, rather than a value assignment. This flag is valid only when the property accepts a reference to an object.

pparams

Pointer to a structure containing an array of arguments, an array of argument dispatch IDs for named

arguments, and counts for number of elements in the arrays.

pvarResult

Pointer to where the result is to be stored, or Null if the caller expects no result. This argument is ignored if DISPATCH_PROPERTYPUT or DISPATCH_PROPERTYPUTREF is specified.

pexcepinfo

Pointer to a structure containing exception information. This structure should be filled in if DISP_E_EXCEPTION is returned.

puArgErr

The index within *rgvarg* of the first argument that has an error. Arguments are stored in *pdispparams->rgvarg* in reverse order, so the first argument is the one with the highest index in the array. This parameter is returned only when the resulting return value is DISP_E_TYPEMISMATCH or DISP_E_PARAMNOTFOUND.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
DISP_E_BADPARAMCOUNT	The number of elements provided in DISPPARAMS is different from the number of arguments accepted by the method or property.
DISP_E_BADVARTYPE	One of the arguments in DISPPARAMS is not a valid variant type.
DISP_E_EXCEPTION	The application needs to raise an exception. In this case, the structure passed in *pexcepinfo* should be filled in.
DISP_E_MEMBERNOTFOUND	The requested member does not exist.
DISP_E_NONAMEDARGS	This implementation of **IDispatch** does not support named arguments.
DISP_E_OVERFLOW	One of the arguments in DISPPARAMS could not be coerced to the specified type.
DISP_E_PARAMNOTFOUND	One of the parameter IDs does not correspond to a parameter on the method. In this case, *puArgErr* is set to the first argument that contains the error.
DISP_E_PARAMNOTOPTIONAL	A required parameter was omitted.
DISP_E_TYPEMISMATCH	One or more of the arguments could not be coerced. The index of the first parameter with the incorrect type within *rgvarg* is returned in *puArgErr*.
E_INVALIDARG	One of the arguments is invalid.
E_OUTOFMEMORY	Insufficient memory to complete the operation.
Other return codes	Any of the **ITypeInfo::Invoke** errors can

also be returned.

Comments

The parameter *_this* is a pointer to an implementation of the interface that is being deferred to. **DispInvoke** builds a stack frame, coerces parameters using standard coercion rules, pushes them on the stack, and then calls the correct member function in the VTBL.

Example

The following code from the Lines sample file Lines.cpp implements **IDispatch::Invoke** using **DispInvoke**. This function uses m_bRaiseException to signal that an error occurred during the **DispInvoke** call.

```
STDMETHODIMP
CLines::Invoke(
     DISPID dispidMember,
     REFIID riid,
     LCID lcid,
     WORD wFlags,
     DISPPARAMS FAR* pdispparams,
     VARIANT FAR* pvarResult,
     EXCEPINFO FAR* pexcepinfo,
     UINT FAR* puArgErr)
{
     return DispInvoke(
     this, m_ptinfo,
     dispidMember, wFlags, pdispparams,
     pvarResult, pexcepinfo, puArgErr);
}
```

See Also

CreateStdDispatch, **IDispatch::Invoke**

Registering the Active Object with API Functions

These functions let you identify a running instance of an object. Because they use the OLE object table (**GetRunningObjectTable**), they also require either Ole32.dll (for 32-bit systems) or Ole2.dll (for 16-bit systems).

When an application is started with the **/Automation** switch, it should initialize its Application object as the active object by calling **RegisterActiveObject** after it initializes OLE.

Applications can also register other top-level objects as the active object. For example, an application that exposes a Document object may want to let ActiveX clients retrieve and modify the currently active document.

For more information about registering the active object, see Chapter 2, "Exposing ActiveX objects." The following table identifies the location of these API functions

Implemented by	Used by	Header file name	Import library name
Oleaut32.dll (32-bit systems) Ole2disp.dll (16-bit systems)	Applications that expose or access programmable objects.	Oleauto.h Dispatch.h	Oleaut32.lib Ole2disp.lib

GetActiveObject Quick Info

HRESULT GetActiveObject(

REFCLSID *rclsid*,
void FAR* *pvreserved*,
IUnknown FAR* FAR* *ppunk*
);

Retrieves a pointer to a running object that has been registered with OLE.

Parameters

rclsid

Pointer to the class ID of the active object from the OLE registration database.

pvreserved

Reserved for future use. Must be Null.

ppunk

On return, a pointer to the requested active object.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
Other return codes	Failure.

RegisterActiveObject Quick Info

HRESULT RegisterActiveObject (

IUnknown FAR* *punk*,
REFCLSID *rclsid*,
DWORD *dwFlags*,
unsigned long FAR* *pdwRegister*
);

Registers an object as the active object for its class.

Parameters

punk

Pointer to the **IUnknown** interface of the active object.

rclsid

Pointer to the class ID of the active object.

dwFlags

Flags controlling registration of the object. Possible values are ACTIVEOBJECT_STRONG and ACTIVEOBJECT_WEAK.

pdwRegister

On return, a pointer to a handle. This handle must be passed to **RevokeActiveObject** to end the object's active status.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
Other return codes	Failure.

Comments

The **RegisterActiveObject** function registers the object to which *punk* points as the active object for the class denoted by *rclsid.* Registration causes the object to be listed in the running object table (ROT) of OLE, a globally accessible lookup table that keeps track of objects that are currently running on the computer. (For more information about the running object table, see the *Microsoft OLE Programmer's Guide and Reference*.) The *dwFlags* parameter specifies the strength or weakness of the registration, which affects the way the object is shut down.

In general, ActiveX objects should behave in the following manner:

- If the object is visible, it should shut down only in response to an explicit user command (such as the **Exit** command on the **File** menu), or to the equivalent command from an ActiveX client (invoking the **Quit** or **Exit** method on the Application object).
- If the object is not visible, it should shut down only when the last external connection to it is gone.

Strong registration performs an **AddRef** on the object, incrementing the reference count of the object (and its associated stub) in the running object table. A strongly registered object must be explicitly revoked from the table with **RevokeActiveObject**. The default is strong registration (ACTIVEOBJECT_STRONG).

Weak registration keeps a pointer to the object in the running object table, but does not increment the reference count. Consequently, when the last external connection to a weakly registered object disappears, OLE releases the object's stub, and the object itself is no longer available.

To ensure the desired behavior, consider not only the default actions of OLE, but also the following:

- Even though code can create an invisible object, the object may become visible at some later time. Once the object is visible, it should remain visible and active until it receives an explicit command to shut down. This can occur after references from the code disappear.
- Other ActiveX clients may be using the object. If so, the code should not force the object to shut down.

To avoid possible conflicts, you should always register ActiveX objects with ACTIVEOBJECT_WEAK, and call **CoLockObjectExternal,** when necessary, to guarantee the object remains active. **CoLockObjectExternal** adds a strong lock, thereby preventing the object's reference count from reaching zero. For detailed information about this function, refer to the *Microsoft OLE Programmer's Guide and Reference*.

Most commonly, objects need to call **CoLockObjectExternal** when they become visible, so they remain active until the user requests the object to shut down.

Quick Info

To shut down correctly, code should follow these steps

1. When the object becomes visible, make the following call to add a lock for the user:

   ```
   CoLockObjectExternal(punk, TRUE, TRUE)
   ```

   The lock remains in effect until a user explicitly requests the object to be shut down, such as with a **Quit** or **Exit** command.
2. When the user requests the object to be shut down, call **CoLockObjectExternal** again to free the lock, as follows:

   ```
   CoLockObjectExternal(punk, FALSE, TRUE)
   ```

3. Call **RevokeActiveObject** to make the object inactive.
4. To end all connections from remote processes, call **CoDisconnectObject** as follows:

   ```
   CoDisconnectObject(punk, 0)
   ```

   This function is described in more detail in the *Microsoft OLE Programmer's Guide and Reference*.

RevokeActiveObject Quick Info

HRESULT RevokeActiveObject (

unsigned long *dwRegister*,
void FAR* *pvreserved*
);

Ends an object's status as active.

Parameters

dwRegister

A handle previously returned by **RegisterActiveObject**.

pvreserved

Reserved for future use. Must be Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
Other return codes	Failure.

IEnumVARIANT Interface

The **IEnumVARIANT** interface provides a method for enumerating a collection of variants, including heterogeneous collections of objects and intrinsic types. Callers of this interface do not need to know the specific type (or types) of the elements in the collection.

Implemented by	Used by	Header file name
Applications that expose collections of objects	Applications that access collections of objects	Oleauto.h (32-bit systems) Dispatch.h (16-bit systems)

The following is the definition that results from expanding the parameterized type **IEnumVARIANT**:

```
interface IEnumVARIANT : IUnknown {
     virtual HRESULT Next(unsigned long celt,
                          VARIANT FAR* rgvar,
                          unsigned long FAR* pceltFetched) = 0;
     virtual HRESULT Skip(unsigned long celt) = 0;
     virtual HRESULT Reset() = 0;
     virtual HRESULT Clone(IEnumVARIANT FAR* FAR* ppenum) = 0;
     };
```

To see how to implement a collection of objects using **IEnumVARIANT,** refer to the file Enumvar.cpp in the Lines sample code.

IEnumVARIANT::Clone

HRESULT IEnumVARIANT::Clone(

IEnumVARIANT FAR* FAR* *ppenum*
);

Creates a copy of the current state of enumeration.

Parameter

ppenum

On return, pointer to the location of the clone enumerator.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

Comments

Using this function, a particular point in the enumeration sequence can be recorded, and then returned to at a later time. The returned enumerator is of the same actual interface as the one that is being cloned.

There is no guarantee that exactly the same set of variants will be enumerated the second time as was enumerated the first. Although an exact duplicate is desirable, the outcome depends on the collection being enumerated. You may find that it is impractical for some collections to maintain this condition (for example, an enumeration of the files in a directory).

Example

The following code implements **IEnumVariant::Clone** for collections in the Lines sample file Enumvar.cpp.

```
STDMETHODIMP
CEnumVariant::Clone(IEnumVARIANT FAR* FAR* ppenum)
{
     CEnumVariant FAR* penum = NULL;
     HRESULT hr;

     *ppenum = NULL;

     hr = CEnumVariant::Create(m_psa, m_cElements, &penum);
     if (FAILED(hr))
          goto error;
     penum->AddRef();
     penum->m_lCurrent = m_lCurrent;

     *ppenum = penum;
     return NOERROR;
```

```
error:
     if (penum)
           penum->Release();
     return hr;
}
```

IEnumVARIANT::Next

HRESULT IEnumVARIANT::Next(

unsigned long *celt*,
VARIANT FAR* *rgvar*,
unsigned long FAR* *pceltFetched*
);

Attempts to get the next *celt* items in the enumeration sequence, and return them through the array pointed to by *rgvar*.

Parameters

celt

The number of elements to be returned.

rgvar

An array of at least size *celt* in which the elements are to be returned.

pceltFetched

Pointer to the number of elements returned in *rgvar*, or Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	The number of elements returned is *celt.*
S_FALSE	The number of elements returned is less than *celt*.

Comments

If fewer than the requested number of elements remain in the sequence, **Next** returns only the remaining elements. The actual number of elements returned is passed through **pceltFetched*, unless it is Null.

Example

The following code implements **IEnumVariant::Next** for collections in the Lines sample file Enumvar.cpp.

```
STDMETHODIMP
CEnumVariant::Next(ULONG cElements, VARIANT FAR* pvar, ULONG FAR*
pcElementFetched)
{
     HRESULT hr;
     ULONG l;
     long l1;
     ULONG l2;

     if (pcElementFetched != NULL)
           *pcElementFetched = 0;
```

```
    for (l=0; l<cElements; l++)
          VariantInit(&pvar[l]);

    // Retrieve the next cElements elements.
    for (l1=m_lCurrent, l2=0; l1<(long)(m_lLBound+m_cElements) &&
          l2<cElements; l1++, l2++)
    {
          hr = SafeArrayGetElement(m_psa, &l1, &pvar[l2]);
          if (FAILED(hr))
                goto error;
    }
    // Set count of elements retrieved.
    if (pcElementFetched != NULL)
          *pcElementFetched = l2;
    m_lCurrent = l1;

    return  (l2 < cElements) ? ResultFromScode(S_FALSE) : NOERROR;

error:
    for (l=0; l<cElements; l++)
          VariantClear(&pvar[l]);
    return hr;
}
```

IEnumVARIANT::Reset

HRESULT IEnumVARIANT::Reset()

Resets the enumeration sequence to the beginning.

Parameter

None

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
S_FALSE	Failure.

Comments

There is no guarantee that exactly the same set of variants will be enumerated the second time as was enumerated the first time. Although an exact duplicate is desirable, the outcome depends on the collection being enumerated. You may find that it is impractical for some collections to maintain this condition (for example, an enumeration of the files in a directory).

Example

The following code implements **IEnumVariant::Reset** for collections in the Lines sample file Enumvar.cpp:

```
STDMETHODIMP
CEnumVariant::Reset()
{
     m_lCurrent = m_lLBound;
     return NOERROR;
}
```

IEnumVARIANT::Skip

HRESULT IEnumVARIANT::Skip(

unsigned long *celt*
);

Attempts to skip over the next *celt* elements in the enumeration sequence.

Parameter

celt

The number of elements to skip.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	The specified number of elements was skipped.
S_FALSE	The end of the sequence was reached before skipping the requested number of elements.

Example

The following code implements **IEnumVariant::Reset** for collections in the Lines sample file Enumvar.cpp.

```
STDMETHODIMP
CEnumVariant::Skip(ULONG cElements)
{
     m_lCurrent += cElements;
     if (m_lCurrent > (long)(m_lLBound+m_cElements))
     {
           m_lCurrent =  m_lLBound+m_cElements;
           return ResultFromScode(S_FALSE);
     }
     else return NOERROR;
}
```

Data Types, Structures, and Enumerations

Each Automation interface has associated data information. This chapter contains information on the following:

- Data types
- Data structures
- Data enumerations

The interfaces discussed are:

- **IDispatch**
- **ITypeInfo**
- **ITypeLib**
- **ITypeComp**

IDispatch Data Types and Structures

The **IDispatch** interface uses the following data types and structures. For more information on the implementation of the **IDispatch** interface, see Chapter 5, "Dispatch Interface and API Functions."

Name	Purpose
BSTR	A length-prefixed string.
CALLCONV	Identifies the calling convention used by a member function.
CURRENCY	Provides a precise data type of monetary data.
DECIMAL	Provides a decimal data type.
DISPID	Identifies a method, property, or argument to **Invoke**.
DISPPARAMS	Contains arguments passed to a method or property.
EXCEPINFO	Describes an error that occurred during **Invoke**.
INTERFACEDATA	Describes the members of an interface.
LCID	Provides locale information for international string comparisons and localized member names.
METHODDATA	Describes a method or property.
PARAMDATA	Describes a parameter to a method.
VARIANT	Describes a VARIANTARG that cannot have the VT_BYREF bit set. Because VT_BYREF is not set, the data of type VARIANT cannot be passed within DISPPARAMS.
VARIANTARG	Describes arguments that may be passed within DISPPARAMS.
VARTYPE	Identifies the available variant types.
VARENUM	Used in VARIANT, TYPEDESC, and OLE (ActiveX) property sets.

BSTR

A length-prefixed string used by Automation data manipulation functions.

```
typedef OLECHAR *BSTR;
```

BSTRs (binary strings) are wide, double-byte (Unicode) strings on 32-bit Windows platforms and narrow, single-byte strings on the Apple® PowerMac™.

For details on the BSTR data type, see Chapter 7, "Conversion and Manipulation Functions."

CALLCONV

Identifies the calling convention used by a member function described in the METHODDATA structure.

```
typedef enum tagCALLCONV {
      CC_CDECL = 1
    , CC_MSCPASCAL
    , CC_PASCAL = CC_MSCPASCAL
    , CC_MACPASCAL
    , CC_STDCALL
    , CC_RESERVED
    , CC_SYSCALL
    , CC_MPWCDECL
    , CC_MPWPASCAL
    , CC_MAX                 // End of enum marker.
} CALLCONV;
```

On 16-bit Windows systems, functions implemented with the CC_CDECL calling convention cannot have a return type of **float** or **double**. This includes functions that return **DATE**, which is a floating-point type.

CURRENCY

A currency number stored as an 8-byte, two's complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This representation provides a range of ±922337203685477.5807. The **CURRENCY** data type is useful for calculations involving money, or for any fixed-point calculation where accuracy is particularly important.

```
typedef CY CURRENCY;
```

The data type is defined as a structure for working with currency more conveniently:

```
typedef struct  tagCY
    {
    LONGLONG int64;
    }CY;

#else
// Real definition that works with the C++ compiler.
typedef union tagCY {
    struct {
#ifdef _MAC
        long          Hi;
        unsigned long Lo;
#else
        unsigned long Lo;
        long          Hi;
#endif
    };
    LONGLONG int64;
} CY;
#endif
#endif
// Size is 8.
typedef CY CURRENCY;
```

DECIMAL

A decimal data type that provides a size and scale for a number (as in coordinates).

See also "Numeric Parsing Functions" in Chapter 7, "Conversion and Manipulation Functions."

```
typedef struct tagDEC
      {
      unsigned short wReserved;
      union {
      struct {
            char sign;
            char scale;
      };
```

DISPID

Used by **IDispatch::Invoke** to identify methods, properties, and arguments.

```
typedef LONG DISPID;
```

The following (dispatch IDs) DISPIDs have special meaning.

DISPID	Description
DISPID_VALUE	The default member for the object. This property or method is invoked when an ActiveX client specifies the object name without a property or method.
DISPID_NEWENUM	The **_NewEnum** property. This special, restricted property is required for collection objects. It returns an enumerator object that supports **IEnumVARIANT,** and should have the **restricted** attribute specified in object description language.
DISPID_EVALUATE	The **Evaluate** method. This method is implicitly invoked when the ActiveX client encloses the arguments in square brackets. For example, the following two lines are equivalent: x.[A1:C1].value = 10 x.Evaluate("A1:C1").value = 10 The **Evaluate** method has the dispatch ID DISPID_EVALUATE.
DISPID_PROPERTYPUT	The parameter that receives the value of an assignment in a PROPERTYPUT.
DISPID_CONSTRUCTOR	The C++ constructor function for the object.
DISPID_DESTRUCTOR	The C++ destructor function for the object.
DISPID_UNKNOWN	The value returned by **IDispatch::GetIDsOfNames** to indicate that a member or parameter name was not found.

Note The reserved dispatch IDs (DISPIDs) are:

DISPID_Name	-800
DISPID_Delete	-801
DISPID_Object	-802
DISPID_Parent	-803

DISPPARAMS

Used by **IDispatch::Invoke** to contain the arguments passed to a method or property. For more information, see "IDispatch::Invoke" in Chapter 5, "Dispatch Interface and API Functions."

```
typedef struct FARSTRUCT tagDISPPARAMS{
VARIANTARG FAR* rgvarg;                  // Array of arguments.
     DISPID FAR* rgdispidNamedArgs;      // Dispatch IDs of named arguments.
     unsigned int cArgs;                     // Number of arguments.
     unsigned int cNamedArgs;                // Number of named arguments.
} DISPPARAMS;
```

EXCEPINFO

Describes an exception that occurred during **IDispatch::Invoke**. For more information on exceptions, see "IDispatch::Invoke" in Chapter 5, "Dispatch Interface and API Functions."

```
typedef struct FARSTRUCT tagEXCEPINFO {
     unsigned short wCode;         // An error code describing the error.
     unsigned short wReserved;
     BSTR bstrSource;                 // Source of the exception.
     BSTR bstrDescription;         // Textual description of the error.
     BSTR bstrHelpFile;               // Help file path.
     unsigned long dwHelpContext;  // Help context ID.
     void FAR* pvReserved;
     // Pointer to function that fills in Help and description info.
     HRESULT (STDAPICALLTYPE FAR* pfnDeferredFillIn)
                (struct tagEXCEPINFO FAR*);
     RETURN VALUE return value;    // A return value describing the error.
} EXCEPINFO, FAR* LPEXCEPINFO;
```

The following table describes the fields of the EXCEPINFO structure.

Field name	Type	Description
wCode	unsigned short	An error code identifying the error. Error codes should be greater than 1000. Either this field or the return value field must be filled in; the other must be set to 0.
wReserved	unsigned short	Reserved; should be set to 0.
bstrSource	BSTR	A textual, human-readable name of the source of the exception. Typically, this is an application name. This field should be filled in by the implementor of **IDispatch**.
BstrDescription	BSTR	A textual, human-readable description of the error intended for the customer. If no description is available, use Null.
bstrHelpFile	BSTR	The fully qualified drive, path, and file name of a Help file with more information about the error. If no Help is available, use Null.
DwHelpContext	unsigned long	The Help context ID of the topic within the Help file. This field should be filled in if and only if the bstrHelpFile field is not Null.
pvReserved	void FAR*	Must be set to Null.
PfnDeferredFillIn	STDAPICALLTYPE	Pointer to a function that takes an EXCEPINFO structure as an argument and returns an HRESULT value. If deferred, fill-in is not desired, this field should

		be set to Null.
scode	SCODE	A return value describing the error. Either this field or wCode (but not both) must be filled in; the other must be set to 0. (16-bit versions only)

Use the *pfnDeferredFillIn* field to allow an object to defer filling in the *bstrDescription*, *bstrHelpFile*, and *dwHelpContext* fields until they are needed. This field might be used, for example, if loading the string for the error is a time-consuming operation. To use deferred fill-in, the object puts a function pointer in this slot and does not fill any of the other fields except *wCode*, which is required.

To get additional information, the caller passes the EXCEPINFO structure back to the *pexcepinfo* callback function, which fills in the additional information. When the ActiveX object and the ActiveX client are in different processes, the ActiveX object calls *pfnDeferredFillIn* before returning to the controller.

INTERFACEDATA

Describes the ActiveX object's properties and methods.

```
typedef struct FARSTRUCT tagINTERFACEDATA {
     METHODDATA FAR* pmethdata;    // Pointer to an array of METHODDATAs.
     unsigned int cMembers;        // Count of members.
} INTERFACEDATA;
```

LCID

Identifies a locale for national language support. Locale information is used for international string comparisons and localized member names. For information on locale IDs, see "Supporting Multiple National Languages" in Chapter 2, "Exposing ACTIVEX Objects."

```
typedef unsigned long LCID;
```

METHODDATA

Used to describe a method or property.

```
typedef struct FARSTRUCT tagMETHODDATA {
     OLECHAR FAR* szName;                  // Member name.
     PARAMDATA FAR* ppdata;          // Pointer to array of PARAMDATAs.
     DISPID dispid;                        // Member ID.
     unsigned int iMeth;                   // Method index.
     CALLCONV cc;                               // Calling convention.
     unsigned int cArgs;                   // Count of arguments.
     unsigned short wFlags;          // Description of whether this is a
                                               // method or a PROPERTYGET,
PROPERTYPUT, or                                          // or
PROPERTYPUTREF.
     VARTYPE vtReturn;               // Return type.
} METHODDATA;
```

The following table describes the fields of the METHODDATA structure.

Name	Type	Description
szName	OLECHAR FAR*	The method name.
Ppdata	PARAMDATA FAR*	The parameters for the method. The first parameter is ppdata[0], and so on.
Dispid	DISPID	The ID of the method, as used in **IDispatch**.
IMeth	unsigned int	The index of the method in the VTBL of the interface. The indexes start with 0.
Cc	CALLCONV	The calling convention. The CDECL and Pascal calling conventions are supported by the dispatch interface creation functions, such as **CreateStdDispatch**.
CArgs	unsigned int	The number of arguments for the method.
WFlags	unsigned short	Flags that indicate whether the method is used for getting or setting a property. The flags are the same as in **IDispatch::Invoke**. DISPATCH_METHOD indicates that this is not used for a property. DISPATCH_PROPERTYGET indicates that the method is used to get a property value. DISPATCH_PROPERTYPUT indicates that the method is used to set the value of a property. DISPATCH_PROPERTYPUTREF indicates that the method is used to make the property refer to a passed-in object.
VtReturn	VARTYPE	Return type for the method.

PARAMDATA

Used to describe a parameter accepted by a method or property.

```
typedef struct FARSTRUCT tagPARAMDATA {
     OLECHAR FAR* szName;          // Parameter name.
     VARTYPE vt;                   // Parameter type.
} PARAMDATA;
```

The following table describes the fields of the PARAMDATA structure.

Name	Type	Description
szName	OLECHAR FAR*	The parameter name. Names should follow standard conventions for programming language access; that is, no embedded spaces or control characters, and 32 or fewer characters. The name should be localized because each type description provides names for a particular locale.
Vt	VARTYPE	The VARTYPE that will be used by the receiver. If more than one parameter type is accepted, VT_VARIANT should be specified.

VARIANT and VARIANTARG

Use VARIANTARG to describe arguments passed within DISPPARAMS, and VARIANT to specify variant data that cannot be passed by reference. The VARIANT type cannot have the VT_BYREF bit set. VARIANTs can be passed by value, even if VARIANTARGs cannot.

```
typedef struct FARSTRUCT tagVARIANT VARIANT;
typedef struct FARSTRUCT tagVARIANT VARIANTARG;

typedef struct tagVARIANT  {
     VARTYPE vt;
     unsigned short wReserved1;
     unsigned short wReserved2;
     unsigned short wReserved3;
     union {
           unsigned char     bVal;             // VT_UI1.
           short             iVal;             // VT_I2    .
           long              lVal;             // VT_I4    .
           float             fltVal;                 // VT_R4    .
           double                  dblVal;                 // VT_R8    .
           VARIANT_BOOL            bool;             // VT_BOOL.
           RETURN VALUE            return value;     // VT_ERROR.
           CY                      cyVal;                  // VT_CY    .
           DATE              date;             // VT_DATE.
           BSTR              bstrVal;                // VT_BSTR.
           IUnknown                FAR* punkVal;     // VT_UNKNOWN.
           IDispatch         FAR* pdispVal;    // VT_DISPATCH.
           SAFEARRAY         FAR* parray;            // VT_ARRAY|*.
           unsigned char     FAR *pbVal;       // VT_BYREF|VT_UI1.
           short             FAR* piVal;       // VT_BYREF|VT_I2.
           long              FAR* plVal;       // VT_BYREF|VT_I4.
           float             FAR* pfltVal;     // VT_BYREF|VT_R4.
           double                  FAR* pdblVal;     // VT_BYREF|VT_R8.
           VARIANT_BOOL            FAR* pbool;       // VT_BYREF|VT_BOOL.
           RETURN VALUE                  FAR* preturn value;     //
VT_BYREF|VT_ERROR.
           CY                      FAR* pcyVal;            // VT_BYREF|
VT_CY.
           DATE              FAR* pdate;       // VT_BYREF|VT_DATE.
           BSTR              FAR* pbstrVal;    // VT_BYREF|VT_BSTR.
           IUnknown FAR*     FAR* ppunkVal;    // VT_BYREF|VT_UNKNOWN.
           IDispatch FAR*    FAR* ppdispVal;   // VT_BYREF|VT_DISPATCH.
           SAFEARRAY FAR*    FAR* parray;            // VT_ARRAY|*.
           VARIANT                 FAR* pvarVal;     // VT_BYREF|VT_VARIANT.
           void              FAR* byref;       // Generic ByRef.
     };
};
```

To simplify extracting values from VARIANTARGs, Automation provides a set of functions for manipulating this type. Use of these functions is strongly recommended to ensure that applications apply consistent coercion rules.

The *vt* value governs the interpretation of the union as follows:

Value	Description

VT_EMPTY	No value was specified. If an optional argument to an Automation method is left blank, do not pass a VARIANT of type VT_EMPTY. Instead, pass a VARIANT of type VT_ERROR with a value of DISP_E_MEMBERNOTFOUND.
VT_EMPTY | VT_BYREF	Not valid.
VT_UI1	An unsigned 1-byte character is stored in *bVal*.
VT_UI1 | VT_BYREF	A reference to an unsigned 1-byte character was passed. A pointer to the value is in *pbVal*.
VT_I2	A 2-byte integer value is stored in *iVal*.
VT_I2 | VT_BYREF	A reference to a 2-byte integer was passed. A pointer to the value is in *piVal*.
VT_I4	A 4-byte integer value is stored in *lVal*.
VT_I4 | VT_BYREF	A reference to a 4-byte integer was passed. A pointer to the value is in *plVal*.
VT_R4	An IEEE 4-byte real value is stored in *fltVal*.
VT_R4 | VT_BYREF	A reference to an IEEE 4-byte real value was passed. A pointer to the value is in *pfltVal*.
VT_R8	An 8-byte IEEE real value is stored in *dblVal*.
VT_R8 | VT_BYREF	A reference to an 8-byte IEEE real value was passed. A pointer to its value is in *pdblVal*.
VT_CY	A currency value was specified. A currency number is stored as an 8-byte, two's complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. The value is in *cyVal*.
VT_CY | VT_BYREF	A reference to a currency value was passed. A pointer to the value is in *pcyVal*.
VT_BSTR	A string was passed; it is stored in *bstrVal*. This pointer must be obtained and freed by the BSTR functions, which are described in Chapter 7, "Conversion and Manipulation Functions."
VT_BSTR | VT_BYREF	A reference to a string was passed. A BSTR* that points to a BSTR is in *pbstrVal*. The referenced pointer must be obtained or freed by the BSTR functions.
VT_NULL	A propagating null value was specified. (This should not be confused with the null pointer.) The null value is used for tri-state logic, as with SQL.
VT_NULL | VT_BYREF	Not valid.
VT_ERROR	An SCODE was specified. The type of the error is specified in *scodee*. Generally, operations on error values should raise an exception or propagate the error to the return

	value, as appropriate.
VT_ERROR | VT_BYREF	A reference to an SCODE was passed. A pointer to the value is in *pscode.*
VT_BOOL	A Boolean (True/False) value was specified. A value of 0xFFFF (all bits 1) indicates True; a value of 0 (all bits 0) indicates False. No other values are valid.
VT_BOOL | VT_BYREF	A reference to a Boolean value. A pointer to the Boolean value is in *pbool*.
VT_DATE	A value denoting a date and time was specified. Dates are represented as double-precision numbers, where midnight, January 1, 1900 is 2.0, January 2, 1900 is 3.0, and so on. The value is passed in *date*. This is the same numbering system used by most spreadsheet programs, although some specify incorrectly that February 29, 1900 existed, and thus set January 1, 1900 to 1.0. The date can be converted to and from an MS-DOS representation using **VariantTimeToDosDateTime**, which is discussed in Chapter 7, "Conversion and Manipulation Functions."
VT_DATE | VT_BYREF	A reference to a date was passed. A pointer to the value is in *pdate*.
VT_DISPATCH	A pointer to an object was specified. The pointer is in *pdispVal*. This object is known only to implement **IDispatch**. The object can be queried as to whether it supports any other desired interface by calling **QueryInterface** on the object. Objects that do not implement **IDispatch** should be passed using VT_UNKNOWN.
VT_DISPATCH | VT_BYREF	A pointer to a pointer to an object was specified. The pointer to the object is stored in the location referred to by *ppdispVal*.
VT_VARIANT	Invalid. VARIANTARGs must be passed by reference.
VT_VARIANT | VT_BYREF	A pointer to another VARIANTARG is passed in *pvarVal*. This referenced VARIANTARG will never have the VT_BYREF bit set in *vt*, so only one level of indirection can ever be present. This value can be used to support languages that allow functions to change the types of variables passed by reference.
VT_UNKNOWN	A pointer to an object that implements the **IUnknown** interface is passed in *punkVal*.
VT_UNKNOWN | VT_BYREF	A pointer to the **IUnknown** interface is passed in *ppunkVal*. The pointer to the interface is stored in the location referred to by *ppunkVal*.
VT_ARRAY | <anything>	An array of data type <anything> was passed. (VT_EMPTY and VT_NULL are invalid types to combine with VT_ARRAY.) The pointer in

pbyrefVal points to an array descriptor, which describes the dimensions, size, and in-memory location of the array. The array descriptor is never accessed directly, but instead is read and modified using the functions described in Chapter 7, "Conversion and Manipulation Functions."

VARTYPE

An enumeration type used in VARIANT, TYPEDESC, OLE property sets, and safe arrays.

The enumeration constants listed in the following VARENUM section are valid in the *vt* field of a VARIANT structure.

```
typedef unsigned short VARTYPE;
enum VARENUM{
     VT_EMPTY          = 0,              // Not specified.
     VT_NULL           = 1,              // Null.
     VT_I2       = 2,              // 2-byte signed int.
     VT_I4       = 3,              // 4-byte signed int.
     VT_R4       = 4,              // 4-byte real.
     VT_R8       = 5,              // 8-byte real.
     VT_CY       = 6,              // Currency.
     VT_DATE           = 7,              // Date.
     VT_BSTR           = 8,              // Binary string.
     VT_DISPATCH = 9,              // IDispatch FAR*.
     VT_ERROR          = 10,             // Return value.
     VT_BOOL           = 11,             // Boolean; True=-1, False=0.
     VT_VARIANT  = 12,             // VARIANT FAR*.
     VT_UNKNOWN  = 13,             // IUnknown FAR*.
     VT_UI1            = 17,             // Unsigned char.

     // Other constants that are not valid in VARIANTs omitted here.

};
     VT_RESERVED = (int) 0x8000
     // By reference, a pointer to the data is passed.
     VT_BYREF    = (int) 0x4000
     VT_ARRAY    = (int) 0x2000    // A safe array of the data is passed.
```

VARENUM

An enumeration type used in VARIANT, TYPEDESC, OLE property sets, and safe arrays.

The following listing identifies the enumerations that apply to each.

```
// VARENUM usage key,
//
// [V] - May appear in a VARIANT.
// [T] - May appear in a TYPEDESC.
// [P] - May appear in an OLE property set.
// [S] - May appear in a safe array.
//
//
VT_EMPTY                    [V]          [P]         // Not specified.
VT_NULL                     [V]                      // SQL-style Null.
VT_I2                  [V][T][P][S]                  // 2-byte signed int.
VT_I4                  [V][T][P][S]                  // 4-byte-signed int.
VT_R4                  [V][T][P][S]                  // 4-byte real.
VT_R8                  [V][T][P][S]                  // 8-byte real.
VT_CY                  [V][T][P][S]                  // Currency.
VT_DATE                     [V][T][P][S]                  // Date.
VT_BSTR                     [V][T][P][S]                  // Automation string.
VT_DISPATCH            [V][T]   [S]                  // IDispatch FAR*.
VT_ERROR                    [V][T]   [S]                  // Return value.
VT_BOOL                     [V][T][P][S]                  // Boolean; True=-1,
False=0.
VT_VARIANT             [V][T][P][S]                  // VARIANT FAR*.
VT_UNKNOWN             [V][T]   [S]                  // IUnknown FAR*.
VT_I1                       [T]                      // Signed char.
VT_UI1                      [V][T]   [S]                  // Unsigned char.
VT_UI2                           [T]                      // Unsigned short.
VT_UI4                           [T]                      // Unsigned short.
VT_I8                       [T][P]                   // Signed 64-bit int.
VT_UI8                           [T]                      // Unsigned 64-bit int.
VT_INT                           [T]                      // Signed machine int.
VT_UINT                          [T]                      // Unsigned machine
int.
VT_VOID                          [T]                      // C-style void.
VT_HRESULT                  [T]
VT_PTR                           [T]                      // Pointer type.
VT_SAFEARRAY                     [T]                      // Use VT_ARRAY in
VARIANT.
VT_CARRAY                   [T]                      // C-style array.
VT_USERDEFINED              [T]                      // User-defined type.
VT_LPSTR                         [T][P]                   // Null-terminated
string.
VT_LPWSTR                   [T][P]                   // Wide null-terminated
string.
VT_FILETIME                      [P]                 // FILETIME.
VT_BLOB                                [P]               // Length-prefixed
bytes.
VT_STREAM                        [P]                 // Name of the stream
follows.
```

```
VT_STORAGE                          [P]         // Name of the storage
follows.
VT_STREAMED_OBJECT                  [P]         // Stream contains an object.
VT_STORED_OBJECT                    [P]         // Storage contains an
object.
VT_BLOB_OBJECT                      [P]         // Blob contains an object.
VT_CF                               [P]         // Clipboard format.
VT_CLSID                                  [P]         // A class ID.
VT_VECTOR                           [P]         // Simple counted array.
VT_ARRAY                                  [V]         // SAFEARRAY*.
VT_BYREF                                  [V]
```

ITypeInfo Data Types

ITypeInfo uses the following structures and enumerations. For more information about the **ITypeInfo** interface , refer to Chapter 9, "Type Description Interfaces."

Name	Purpose
ARRAYDESC	Array description referenced by TYPEDESC, containing the element type, dimension count, and a variable-length array.
ELEMDESC	Includes the type description and process-transfer information for a variable, a function, or a function parameter.
FUNCDESC	Describes a function.
FUNCFLAGS	Enumeration containing constants that are used to define properties of a function.
FUNCKIND	Enumeration for defining whether a function is accessd as a virtual, pure virtual, nonvirtual, static, or through **IDispatch**.
HREFTYPE	A handle identifying a type description.
PARAMDESC	Contains information needed for transferring a structure element, parameter, or function return value between processes.
PARAMFLAGS	Identifies whether the parameter is the return value of a member, the local ID of a client, and passes information from caller to callee, or callee to caller.
IMPLTYPEFLAGS	The interface or dispinterface represents the default for the source or sink. This member of a coclass is called rather than implemented. The member should not be displayed or programmable by users, or sinks receive events through the VTBL.
INVOKEKIND	Defines how a function is called and invoked. Also passed to **IDispatch::Invoke**.
MEMBERID	Identifies the member in a type description. For **IDispatch** interfaces, this is the same as a dispatch ID.
TYPEATTR	Contains attributes of **ITypeInfo**.
TYPEDESC	Describes the type of a variable, the return type of a function, or the type of a function parameter.

TYPEFLAGS	Defines the properties and attributes of a type description.
TYPEKIND	Defines properties of a type.
VARDESC	Describes a variable, constant, or data member.
VARFLAGS	Used to set attributes of a variable.
VARKIND	Defines the kind of variable.

ARRAYDESC

Contained within the TYPEDESC, which describes the type of the array's elements, and information about the array's dimensions. It is defined as follows:

```
typedef struct tagARRAYDESC{
     TYPEDESC tdescElem;                    // Element type.
     unsigned short cDims;                // Dimension count.
     SAFEARRAYBOUND rgbounds[1];          // Variable length array containing
                                                 // one element for each
dimension.
} ARRAYDESC;
```

ELEMDESC

Includes the type description and process-transfer information for a variable, a function, or a function parameter. It is defined as follows:

```
typedef struct tagELEMDESC{
     TYPEDESC tdesc;               // Type of the element.
     PARAMDESC idldesc;            // Information needed for transferring
the
                                         // element between processes.
     }  ELEMDESC;
```

FUNCDESC

Describes a function, and is defined as follows:

```
typedef struct tagFUNCDESC {
     MEMBERID memid;                                  // Function member ID.
     RETURN VALUE FAR* lprgreturn value; // Legal return values for the
                                                        // function.
     ELEMDESC FAR* lprgelemdescParam;          // Array of parameter types.
     FUNCKIND funckind;                               // Specifies whether
the                                                                    //
function is virtual, static,
           // or dispatch-only.
     INVOKEKIND invkind;           // Invocation kind. Indicates if this is
a                                        // property function, and if so,
what kind.
     CALLCONV callconv;            // Specifies the function's calling
                                        // convention.
     short cParams;                // Count of total number of parameters.
     short cParamsOpt;       // Count of optional parameters (detailed
                                        // description follows).
     short oVft;                   // For FUNC_VIRTUAL, specifies the offset
in                                      // the VTBL.
     short cReturn values;   // Count of permitted return values.
     ELEMDESC elemdescFunc;  // Contains the return type of the function.
     unsigned short wFuncFlags;    // Definition of flags follows.
}    FUNCDESC;
```

The *cParams* field specifies the total number of required and optional parameters.

The *cParamsOpt* field specifies the form of optional parameters accepted by the function, as follows:

- A value of 0 specifies that no optional arguments are supported.
- A value of -1 specifies that the method's last parameter is a pointer to a safe array of variants. Any number of variant arguments greater than *cParams* -1 must be packaged by the caller into a safe array and passed as the final parameter. This array of optional parameters must be freed by the caller after control is returned from the call.
- Any other number indicates that the last *n* parameters of the function are variants and do not need to be specified by the caller explicitly. The parameters left unspecified should be filled in by the compiler or interpreter as variants of type VT_ERROR with the value DISP_E_PARAMNOTFOUND.

The fields *cReturn values* and *lprgreturn value* store the count and the set of errors that a function can return. If *cReturn values* = -1, then the set of errors is unknown. If *cReturn values* = -1, or if *cReturn values* = 0, then *lprgreturn value* is undefined.

FUNCFLAGS

Defined as follows:

```
typedef enum tagFUNCFLAGS {
     FUNCFLAG_FRESTRICTED =  1
           , FUNCFLAG_FSOURCE =    0x2
           , FUNCFLAG_FBINDABLE =  0x4
           , FUNCFLAG_FREQUESTEDIT =     0x8
           , FUNCFLAG_FDISPLAYBIND =     0x10
           , FUNCFLAG_FDEFAULTBIND =     0x20
           , FUNCFLAG_FHIDDEN =    0x40
           , FUNCFLAG_FRESTRICTED =                  0x80
           , FUNCFLAG_FDEFAULTCOLLELEM=        0x100
           , FUNCFLAG_FUIDEFAULT =                   0x200
           , FUNCFLAG_FNONBROWSABLE =                0x400
           , FUNCFLAG_FREPLACEABLE =                 0x800
} FUNCFLAGS;
```

Value	Description
FUNCFLAG_FRESTRICTED	The function should not be accessible from macro languages. This flag is intended for system-level functions or functions that type browsers should not display.
FUNCFLAG_FSOURCE	The function returns an object that is a source of events.
FUNCFLAG_FBINDABLE	The function that supports data binding.
FUNCFLAG_FREQUESTEDIT	
FUNCFLAG_FDISPLAYBIND	The function that is displayed to the user as bindable. FUNC_FBINDABLE must also be set.
FUNCFLAG_FDEFAULTBIND	The function that best represents the object. Only one function in a type information can have this attribute.
FUNCFLAG_FHIDDEN	The function should not be displayed to the user, although it exists and is bindable.
FUNCFLAG_FDEFAULTCOLLELEM	Permits an optimization in which the compiler looks for a member named "xyz" on the type of "abc". If such a member is found and is flagged as an accessor function for an element of the default collection, then a call is generated to that member function. Permitted on members in dispinterfaces and interfaces; not permitted on modules. For more information, refer to "defaultcollelem" in Chapter 8, "Type Libraries and the Object Description Language."

FUNCFLAG_FUIDEFAULT	The type information member is the default member for display in the user interface.
FUNCFLAG_FNONBROWSABLE	The property appears in an object browser, but not in a properties browser.
FUNCFLAG_FREPLACEABLE	Tags the interface as having default behaviors.

Note FUNCFLAG_FHIDDEN means that the property should never be shown in object browsers, property browsers, and so on. This function is useful for removing items from an object model. Code can bind to the member, but the user will never know that the member exists.FUNCFLAG_FNONBROWSABLE means that the property should not be displayed in a properties browser. It is used in cicrcumstances in which an error would occur if the property were shown in a properties browser.

Examples

- The **IsSelected** property for a control. Setting it to False would confuse a user if the properties browser was focus-oriented.
- Properties that take a long time to evaluate (for example, a **Count** property for a database object). The time to evaluate might take longer than a user is willing to wait.
- Properties that have side effects.

FUNCFLAG_FRESRICTED means that macro-oriented programmers should not be allowed to access this member. These members are usually treated as _FHIDDEN by tools such as Visual Basic, with the main difference being that code cannot bind to those members.

FUNCKIND

The FUNCKIND enumeration is defined as follows:

```
typedef enum tagFUNCKIND {
     FUNC_VIRTUAL,
     FUNC_PUREVIRTUAL,
     FUNC_NONVIRTUAL,
     FUNC_STATIC,
     FUNC_DISPATCH,
} FUNCKIND;
```

Value	Description
FUNC_PUREVIRTUAL	The function is accessed through the virtual function table, and takes an implicit *this* pointer.
FUNC_VIRTUAL	The function is accessed the same as PUREVIRTUAL, except the function has an implementation.
FUNC_NONVIRTUAL	The function is accessed by static address and takes an implicit *this* pointer.
FUNC_STATIC	The function is accessed by static address and does not take an implicit *this* pointer.
FUNC_DISPATCH	The function can be accessed only through **IDispatch**.

HREFTYPE

A handle that identifies a type description.

```
typedef unsigned long HREFTYPE;
```

IMPLTYPEFLAGS

Defined as follows:

```
#define IMPLTYPEFLAG_FDEFAULT0x1
#define IMPLTYPEFLAG_FSOURCE       0x2
#define IMPLTYPEFLAG_FRESTRICTED   0x4
#define IMPLTYPEFLAG_FDEFAULTVTABLE0x800
```

Value	Description
IMPLTYPEFLAG_FDEFAULT	The interface or dispinterface represents the default for the source or sink.
IMPLTYPEFLAG_FSOURCE	This member of a coclass is called rather than implemented.
IMPLTYPEFLAG_FRESTRICTED	The member should not be displayed or programmable by users.
IMPLTYPEFLAG_FDEFAULTVTABLE	Sinks receive events through the VTBL.

INVOKEKIND

Defined as follows:

```
typedef enum tagINVOKEKIND {
      INVOKE_FUNC = DISPATCH_METHOD,
      INVOKE_PROPERTYGET = DISPATCH_PROPERTYGET,
      INVOKE_PROPERTYPUT = DISPATCH_PROPERTYPUT,
      INVOKE_PROPERTYPUTREF = DISPATCH_PROPERTYPUTREF
} INVOKEKIND;
```

Value	Description
INVOKE_FUNC	The member is called using a normal function invocation syntax.
INVOKE_PROPERTYGET	The function is invoked using a normal property-access syntax.
INVOKE_PROPERTYPUT	The function is invoked using a property value assignment syntax. Syntactically, a typical programming language might represent changing a property in the same way as assignment. For example: `object.property = value`.
INVOKE_PROPERTYPUTREF	The function is invoked using a property reference assignment syntax.

In C, value assignment is written as `*pobj1 = *pobj2`, while reference assignment is written as `pobj1 = pobj2`. Other languages have other syntactic conventions. A property or data member can support only a value assignment, a reference assignment, or both. For a detailed description of property functions, see Chapter 5, "Dispatch Interface and API Functions." The INVOKEKIND enumeration constants are the same constants that are passed to **IDispatch::Invoke** to specify the way in which a function is invoked.

MEMBERID

Identifies the member in a type description. For **IDispatch** interfaces, this is the same as DISPID.

```
typedef DISPID MEMBERID;
```

This is a 32-bit integral value in the following format.

Bits	Value
0 - 15	Offset. Any value is permissible.
16 - 21	The nesting level of this type information in the inheritance hierarchy. For example: `interface mydisp : IDispatch` The nesting level of **IUnknown** is 0, **IDispatch** is 1, and MyDisp is 2.
22 - 25	Reserved. Must be zero.
26 - 28	Value of the dispatch ID.
29	True if this is the member ID for a FUNCDESC; otherwise False.
30 - 31	Must be 01.

Negative IDs are reserved for use by Automation.

PARAMDESC

Contains information needed for transferring a structure element, parameter, or function return value between processes. It is defined as follows:

```
typedef struct FARSTRUCT tagPARAMDESC {
     unsigned long lpVarValue;
     unsigned short wPARAMFlags;
} PARAMDESC
```

The *lpVarValue* field contains a pointer to a VARIANT that describes the default value for this parameter, if the PARAMFLAG_FOPT and PARAMFLAG_FHASDEFAULT bit of *wParamFlags* is set.l.

PARAMFLAGS

Defined as follows:

```
#define PARAMFLAG_NONE       0
#define PARAMFLAG_FIN        0x1
#define PARAMFLAG_FOUT       0x2
#define PARAMFLAG_FLCID      0x4
#define PARAMFLAG_FRETVAL    0x8
#define PARAMFLAG_FOPT       0x10
#define PARAMFLAG_FHASDEFAULT0x20
```

Value	Description
PARAMFLAG_NONE	Whether the parameter passes or receives information is unspecified. **IDispatch** interfaces can use this flag.
PARAMFLAG_FIN	Parameter passes information from the caller to the callee.
PARAMFLAG_FOUT	Parameter returns information from the callee to the caller.
PARAMFLAG_FLCID	Parameter is the locale ID of a client application.
PARAMFLAG_FRETVAL	Parameter is the return value of the member.
PARAMFLAG_FOPT	Parameter is optional. The *lpVarValue* field contains a pointer to a VARIANT describing the default value for this parameter, if the PARAMFLAG_FOPT and PARAMFLAG_FHASDEFAULT bit of *wParamFlags* is set.l.
PARAMFLAG_FHASDEFAULT	Parameter has default behaviors defined. The *lpVarValue* field contains a pointer to a VARIANT that describes the default value for this parameter, if the PARAMFLAG_FOPT and PARAMFLAG_FHASDEFAULT bit of wParamFlags is set.l.

TYPEATTR

Contains attributes of an **ITypeInfo**, and is defined as follows:

```
typedef struct FARSTRUCT tagTYPEATTR {
     GUID guid;                          // The GUID of the type
information.
     LCID lcid;                          // Locale of member names and doc
                                               // strings.
     unsigned long dwReserved;
     MEMBERID memidConstructor;     // ID of constructor, or MEMBERID_NIL if
                                               // none.
     MEMBERID memidDestructor;      // ID of destructor, or MEMBERID_NIL if
                                               // none.
```

```
    OLECHAR FAR* lpstrSchema;     // Reserved for future use.
    unsigned long cbSizeInstance;// The size of an instance of
                                          // this type.
    TYPEKIND typekind;                // The kind of type this
information
                                          // describes.
    unsigned short cFuncs;        // Number of functions.
    unsigned short cVars;         // Number of variables/data members.
    unsigned short cImplTypes;    // Number of implemented interfaces.
    unsigned short cbSizeVft;     // The size of this type's VTBL.
    unsigned short cbAlignment;   // Byte alignment for an instance
                                          // of this type.
    unsigned short wTypeFlags;
    unsigned short wMajorVerNum;  // Major version number.
    unsigned short wMinorVerNum;  // Minor version number.
    TYPEDESC tdescAlias;              // If TypeKind == TKIND_ALIAS,
                                          // specifies the type for
which
                                          // this type is an alias.
    PARAMDESC paramdescType;          // IDL attributes of the
                                          // described type.
} TYPEATTR, FAR* LPTYPEATTR;
```

The *cbAlignment* field indicates how addresses are aligned. A value of 0 indicates alignment on the 64K boundary; 1 indicates no special alignment. For other values, *n* indicates aligned on byte *n*.

TYPEDESC

Describes the type of a variable, the return type of a function, or the type of a function parameter. It is defined as follows:

```
typedef struct FARSTRUCT tagTYPEDESC {
     union {
           // VT_PTR|VT_SAFEAEEAY, the pointed-at type.
     struct FARSTRUCT tagTYPEDESC FAR* lptdesc;

           // VT_CARRAY.
     struct FARSTRUCT tagARRAYDESC FAR* lpadesc;

           // VT_USERDEFINED is used to get type information for a
           // user-defined type.
     HREFTYPE hreftype;

     }UNION_NAME(u);
     VARTYPE vt;
} TYPEDESC;
```

If the variable is VT_SAFEARRAY or VT_PTR, the union portion of the TYPEDESC contains a pointer to a TYPEDESC that specifies the element type.

TYPEFLAGS

The TYPEFLAGS enumeration is defined as follows:

```
typedef enum tagTYPEFLAGS {
     TYPEFLAG_FAPPOBJECT =          0x01
     , TYPEFLAG_FCANCREATE =        0x02
     , TYPEFLAG_FLICENSED =         0x04
     , TYPEFLAG_FPREDECLID =        0x08
     , TYPEFLAG_FHIDDEN =                 0x10
     , TYPEFLAG_FCONTROL =          0x20
     , TYPEFLAG_FDUAL =                   0x40
     , TYPEFLAG_FNONEXTENSIBLE =    0x80
     , TYPEFLAG_FOLEAUTOMATION =    0x100
     , TYPEFLAG_FAGGREGATABLE =     0x400
     , TYPEFLAG_FREPLACEABLE =      0x800
     , TYPEFLAG_FDISPATCHABLE =     0x1000
} TYPEFLAGS;
```

Value	Description
TYPEFLAG_FAPPOBJECT	A type description that describes an Application object.
TYPEFLAG_FCANCREATE	Instances of the type can be created by **ITypeInfo::CreateInstance**.
TYPEFLAG_FLICENSED	The type is licensed.
TYPEFLAG_FPREDECLID	The type is predefined. The client application should automatically create a single instance of the object that has this attribute. The name of the variable that points to the object is the same as the class name of the object.
TYPEFLAG_FHIDDEN	The type should not be displayed to browsers.
TYPEFLAG_FCONTROL	The type is a control from which other types will be derived, and should not be displayed to users.
TYPEFLAG_FDUAL	The types in the interface derive from **IDispatch** and are fully compatible with Automation. Not allowed on dispinterfaces (dispatch interfaces).
TYPEFLAG_FNONEXTENSIBLE	The interface cannot add members at run time.
TYPEFLAG_FOLEAUTOMATION	The types used in the interface are fully compatible with Automation, and may be displayed in an object browser. Setting **dual** on an interface sets this flag in addition to TYPEFLAG_FDUAL. Not allowed on dispinterfaces.
TYPEFLAG_FAGGREGATABLE	The class supports aggregation.
TYPEFLAG_FREPLACEABLE	The object supports

	IConnectionPointWithDefault, and has default behaviors.
TYPEFLAG_FDISPATCHABLE	Indicates that the interface derives from **IDispatch**, either directly or indirectly. This flag is computed. There is no Object Description Language for the flag.

TYPEFLAG_FAPPOBJECT can be used on type descriptions with TypeKind = TKIND_COCLASS, and indicates that the type description specifies an Application object.

Members of the Application object are globally accessible. The **Bind** method of the **ITypeComp** instance associated with the library binds to the members of an Application object, just as it does for type descriptions that have TypeKind = TKIND_MODULE.

The type description implicitly defines a global variable with the same name and type described by the type description. This variable is also globally accessible. When **Bind** is passed the name of an Application object, a VARDESC is returned, which describes the implicit variable. The ID of the implicitly created variable is always ID_DEFAULTINST.

The **ITypeInfo::CreateInstance** function of an Application object type description is called, and then it uses **GetActiveObject** to retrieve the Application object. If **GetActiveObject** fails because the application is not running, then **CreateInstance** calls **CoCreateInstance**, which should start the application.

When TYPEFLAG_FCANCREATE is True, **ITypeInfo::CreateInstance** can create an instance of the type. This is currently true only for component object classes for which a globally unique ID has been specified.

TYPEKIND

Defined as follows:

```
typedef enum tagTYPEKIND {
       TKIND_ENUM = 0
     , TKIND_RECORD
     , TKIND_MODULE
     , TKIND_INTERFACE
     , TKIND_DISPATCH
     , TKIND_COCLASS
     , TKIND_ALIAS
     , TKIND_UNION
     , TKIND_MAX
} TYPEKIND;
```

Value	Description
TKIND_ALIAS	A type that is an alias for another type.
TKIND_COCLASS	A set of implemented component object interfaces.
TKIND_DISPATCH	A set of methods and properties that are accessible through **IDispatch::Invoke**. By default, dual interfaces return TKIND_DISPATCH.
TKIND_ENUM	A set of enumerators.
TKIND_INTERFACE	A type that has virtual functions, all of which are pure.
TKIND_MODULE	A module that can only have static functions and data (for example, a DLL).
TKIND_RECORD	A structure with no methods.
TKIND_UNION	A union, all of whose members have an offset of zero.
TKIND_MAX	End of ENUM marker.

VARDESC

Describes a variable, constant, or data member. It is defined as follows:

```
typedef struct FARSTRUCT tagVARDESC {
     MEMBERID memid;
     OLECHAR FAR* lpstrSchema;     // Reserved for future use.
     union {
                                                  // VAR_PERINSTANCE, the
offset of this
                                                  // variable within the
instance.
     unsigned long oInst;

                                                  // VAR_CONST, the value of
the constant.
     VARIANT FAR* lpvarValue;

     } UNION_NAME(u);
     ELEMDESC elemdescVar;
     unsigned short wVarFlags;
     VARKIND varkind;
} VARDESC
```

VARFLAGS

Defined as follows:

```
typedef enum tagVARFLAGS {
            VARFLAG_FREADONLY =                     0x1
          , VARFLAG_FSOURCE =                     0x2
          , VARFLAG_FBINDABLE =                   0x4
          , VARFLAG_FREQUESTEDIT =                0x8
          , VARFLAG_FDISPLAYBIND =                0x10
          , VARFLAG_FDEFAULTBIND =                0x20
          , VARFLAG_FHIDDEN =                     0x40
          , VARFLAG_FRESTRICTED =                 0x80
          , VARFLAG_FDEFAULTCOLLELEM=       0x100
          , VARFLAG_FUIDEFAULT =                      0x200
          , VARFLAG_FNONBROWSABLE =               0x400
          , VARFLAG_FREPLACEABLE =                0x800
} VARFLAGS;
```

Value	Description
VARFLAG_FREADONLY	Assignment to the variable should not be allowed.
VARFLAG_FSOURCE	The variable returns an object that is a source of events.
VARFLAG_FBINDABLE	The variable supports data binding.
VARFLAG_FREQUESTEDIT	
VARFLAG_FDISPLAYBIND	The variable is displayed to the user as bindable. VARFLAG_FBINDABLE must also be set.
VARFLAG_FDEFAULTBIND	The variable is the single property that best represents the object. Only one variable in type information can have this attribute.
VARFLAG_FHIDDEN	The variable should not be displayed to the user in a browser, although it exists and is bindable.
VARFLAG_FRESTRICTED	The variable should not be accessible from macro languages. This flag is intended for system-level variables or variables that you do not want type browsers to display.
VARFLAG_FDEFAULTCOLLELEM	Permits an optimization in which the compiler looks for a member named "xyz" on the type of abc. If such a member is found and is flagged as an accessor function for an element of the default collection, then a call is

	generated to that member function. Permitted on members in dispinterfaces and interfaces; not permitted on modules.
VARFLAG_FUIDEFAULT	The variable is the default display in the user interface.
VARFLAG_FNONBROWSABLE	The variable appears in an object browser, but not in a properties browser.
VARFLAG_FREPLACEABLE	Tags the interface as having default behaviors.
VARFLAG_FIMMEDIATEBIND	The variable is mapped as individual bindable properties.

VARKIND

Defined as follows:

```
typedef enum tagVARKIND {
     VAR_PERINSTANCE,
     VAR_STATIC,
     VAR_CONST,
     VAR_DISPATCH
} VARKIND;
```

Value	Description
VAR_PERINSTANCE	The variable is a field or member of the type. It exists at a fixed offset within each instance of the type.
VAR_STATIC	There is only one instance of the variable.
VAR_CONST	The VARDESC describes a symbolic constant. There is no memory associated with it.
VAR_DISPATCH	The variable can only be accessed through **IDispatch::Invoke**.

ITypeLib Structures and Enumerations

The type building interfaces use the following structures and enumerations.

LIBFLAGS

Defines flags that apply to type libraries. It is defined as follows:

```
typedef enum tagLIBFLAGS {
       LIBFLAG_FRESTRICTED = 0x01
     , LIBFLAG_FCONTROL = 0x02
     , LIBFLAG_FHIDDEN = 0x04
} LIBFLAGS;
```

Value	Description
LIBFLAG_FCONTROL	The type library describes controls, and should not be displayed in type browsers intended for nonvisual objects.
LIBFLAG_FRESTRICTED	The type library is restricted, and should not be displayed to users.
LIBFLAG_FHIDDEN	The type library should not be displayed to users, although its use is not restricted. Should be used by controls. Hosts should create a new type library that wraps the control with extended properties.

REGKIND

Control how a type library is registered

```
typedef enum tagREGKIND{
    REGKIND_DEFAULT,
    REGKIND_REGISTER,
    REGKIND_NONE
} REGKIND;
```

Value	Description
REGKIND_DEFAULT	Use default register behavior
REGKIND_REGISTER	Registered type
REGKIND_NONE	Not a registered type

SYSKIND

Identifies the target operating system platform. It is defined as follows:

```
typedef enum tagSYSKIND {
     SYS_WIN16,
     SYS_WIN32,
     SYS_MAC
} SYSKIND;
```

Value	Description
SYS_WIN16	The target operating system for the type library is 16-bit Windows systems. By default, data members are packed.
SYS_WIN32	The target operating system for the type library is 32-bit Windows systems. By default, data members are naturally aligned (for example, 2-byte integers are aligned on even-byte boundaries; 4-byte integers are aligned on quad-word boundaries, and so on).
SYS_MAC	The target operating system for the type library is Apple Macintosh. By default, all data members are aligned on even-byte boundaries.

TLIBATTR

Contains information about a type library. Information from this structure is used to identify the type library and to provide national language support for member names. It is defined as follows:

```
typedef struct FARSTRUCT tagTLIBATTR {
    GUID guid;                      // Unique ID of the library.
    LCID lcid;                      // Language/locale of the library.
    SYSKIND syskind;                // Target hardware platform.
    unsigned short wMajorVerNum;    // Major version number.
    unsigned short wMinorVerNum;    // Minor version number.
    unsigned short wLibFlags;       // Library flags.
} TLIBATTR, FAR * LPTLIBATTR;
```

For more information on national language support, see "Supporting Multiple National Languages" in Chapter 2, "Exposing ActiveX objects," and refer to the National Language Support API reference material in the Windows NT® documentation.

ITypeComp Structures and Enumerations

The **ITypeComp** interface uses the following structures and enumerations:

BINDPTR

A union containing a pointer to a FUNCDESC, VARDESC, or an **ITypeComp** interface. It is defined as follows:

```
typedef union tagBINDPTR {
     FUNCDESC FAR* lpfuncdesc;
     VARDESC FAR* lpvardesc;
     ITypeComp FAR* lptcomp;
} BINDPTR;
```

DESCKIND

Identifies the type description being bound to, and is defined as follows:

```
typedef enum tagDESCKIND {
     DESCKIND_NONE,
     DESCKIND_FUNCDESC,
     DESCKIND_VARDESC,
     DESCKIND_TYPECOMP,
     DESCKIND_IMPLICITAPPOBJ
} DESCKIND;
```

Comments

Value	Description
DESCKIND_NONE	No match was found.
DESCKIND_FUNCDESC	A FUNCDESC was returned.
DESCKIND_VARDESC	A VARDESC was returned.
DESCKIND_TYPECOMP	A TYPECOMP was returned.
DESCKIND_IMPLICITAPPOBJ	An IMPLICITAPPOBJ was returned.

Conversion and Manipulation Functions

Data manipulation and conversion functions access and manipulate the array, string, and variant data types used by Automation. This chapter contains information about the following functions:

- Array manipulation
- String manipulation
- Variant manipulation
- Data type conversion
- BSTR and vector conversion
- Date and time conversion

You can locate all of the data functions and data types in the following files.

Implemented by	Used by	Header file name	Import library file name
Oleaut32.dll (32-bit systems) Ole2disp.dll (16-bit systems)	Applications that expose or access programmable objects.	Oleauto.h. Dispatch.h	Oleauto32.lib Ole2disp.lib

Overview of Functions

The data manipulation functions are summarized in the following table.

Category	Function name	Purpose
Array manipulation	SafeArrayAccessData	Increments the lock count of an array and returns a pointer to array data.
	SafeArrayAllocData	Allocates memory for a safe array based on a descriptor created with SafeArrayAllocDescriptor.
	SafeArrayAllocDescriptor	Allocates memory for a safe array descriptor.
	SafeArrayCopy	Copies an existing array.
	SafeArrayCopyData	Copies a source array to a target array after releasing source resources.
	SafeArrayCreate	Creates a new array descriptor.
	SafeArrayCreateVector	Creates a one-dimensional array whose lower bound is always zero.
	SafeArrayDestroy	Destroys an array descriptor.
	SafeArrayDestroyData	Frees memory used by the data elements in a safe array.
	SafeArrayDestroyDescriptor	Frees memory used by a safe array descriptor.
	SafeArrayGetDim	Returns the number of dimensions in an array.
	SafeArrayGetElement	Retrieves an element of an array.
	SafeArrayGetElemsize	Returns the size of an element.
	SafeArrayGetLBound	Retrieves the lower bound for a given dimension.
	SafeArrayGetUBound	Retrieves the upper bound for a given dimension.
	SafeArrayLock	Increments the lock count of an array.
	SafeArrayPtrOfIndex	Returns a pointer to an array element.
	SafeArrayPutElement	Assigns an element into an array.
	SafeArrayRedim	Resizes a safe array.
	SafeArrayUnaccessData	Frees a pointer to array data and decrements the lock count of the array.
	SafeArrayUnlock	Decrements the lock count of an array.

String manipulation	**SysAllocString**	Creates and initializes a string.
	SysAllocStringByteLen	Creates a zero-terminated string of a specified length (32-bit only).
	SysAllocStringLen	Creates a string of a specified length.
	SysFreeString	Frees a previously created string.
	SysReAllocString	Changes the size and value of a string.
	SysReAllocStringLen	Changes the size of an existing string.
	SysStringByteLen	Returns the length of a string in bytes (32-bit only).
	SysStringLen	Returns the length of a string.
Variant manipulation	**VariantChangeType**	Converts a variant to another type.
	VariantChangeTypeEx	Converts a variant to another type, using a locale ID.
	VariantClear	Releases resources and sets a variant to VT_EMPTY.
	VariantCopy	Copies a variant.
	VariantCopyInd	Copies variants that may contain a pointer.
	VariantInit	Initializes a variant.
Data type conversion	**VariantChangeType** **VariantChangeTypeEx**	Converts specific types of variants to other variant types.
BSTR and vector conversion	**VectorFromBstr**	Returns a vector, assigning each character in the BSTR to an element of the vector.
	BstrFromVector	Returns a BSTR, assigning each element of the vector to a character in the BSTR.
Time and Date conversion	**DosDateTimeToVariantTime**	Converts MS-DOS date and time representations to a variant time.
	VariantTimeToDosDateTime	Converts a variant time to MS-DOS date and time representations.
	VariantTimeToSystemTime	Converts a variant time to system date and time representations.
	SystemTimeToVariantTime	Converts system date and time representations to a variant time.

Array Manipulation API Functions

The arrays passed by **IDispatch::Invoke** within VARIANTARGs are called *safe arrays*. Safe arrays contain information about the number of dimensions and bounds within them. When an array is an argument or the return value of a function, the *parray* field of VARIANTARG points to an array descriptor. Do not access this array descriptor directly, unless you are creating arrays containing elements with nonvariant data types. Instead, use the functions **SafeArrayAccessData** and **SafeArrayUnaccessData** to access the data.

The base type of the array is indicated by VT_ tag | VT_ARRAY. The data referenced by an array descriptor is stored in column-major order, which is the same ordering scheme used by Visual Basic and FORTRAN, but different from C and Pascal. *Column-major order* is when the left-most dimension (as specified in a programming language syntax) changes first.

The following sections define the safe array descriptor, along with the functions you use when accessing the data in the descriptor and the array.

SAFEARRAY Data Type

The definition for a safe array varies, depending on the target operating system platform. On 32-bit Windows systems, both the *cbElements* and *cLocks* parameters are **unsigned long** integers, and the *handle* parameter is omitted. On 16-bit Windows systems, *cbElements* and *cLocks* are **unsigned short** integers The *handle* parameter is retained for compatibility with earlier software. For example:

```
typedef struct FARSTRUCT tagSAFEARRAY {
     unsigned short cDims;         // Count of dimensions in this array.
     unsigned short fFeatures;     // Flags used by the SafeArray
                                             // routines documented below.
#if defined(WIN32)
     unsigned long cbElements;     // Size of an element of the array.
                                             // Does not include size of
                                             // pointed-to data.
     unsigned long cLocks;         // Number of times the array has been
                                             // locked without
corresponding unlock.
#else
     unsigned short cbElements;
     unsigned short cLocks;
     unsigned long handle;         // Unused but kept for compatibility.
#endif
     void HUGEP* pvData;                     // Pointer to the data.
     SAFEARRAYBOUND rgsabound[1];        // One bound for each dimension.
} SAFEARRAY;
```

The array rgsabound is stored with the left-most dimension in rgsabound[0] and the right-most dimension in rgsabound[*cDims* - 1]. If an array was specified in a C-like syntax as a [2][5], it would have two elements in the rgsabound vector. Element 0 has an lLbound of 0 and a cElements of 2. Element 1 has an lLbound of 0 and a cElements of 5.

The fFeatures flags describe attributes of an array that can affect how the array is released. This allows freeing the array without referencing its containing variant. The bits are accessed using the following constants:

```
#define FADF_AUTO      0x0001      // Array is allocated on the stack.
#define FADF_STATIC          0x0002      // Array is statically allocated.
#define FADF_EMBEDDED  0x0004      // Array is embedded in a structure.
#define FADF_FIXEDSIZE 0x0010      // Array may not be resized or
                                             // reallocated.
#define FADF_BSTR      0x0100      // An array of BSTRs.
#define FADF_UNKNOWN         0x0200      // An array of IUnknown*.
#define FADF_DISPATCH  0x0400      // An array of IDispatch*.
#define FADF_VARIANT         0x0800      // An array of VARIANTs.
#define FADF_RESERVED  0xF0E8      // Bits reserved for future use.
```

SAFEARRAYBOUND Structure

Represents the bounds of one dimension of the array. The lower bound of the dimension is represented by lLbound, and cElements represents the number of elements in the dimension. The structure is defined as follows:

```
typedef struct tagSAFEARRAYBOUND {
     unsigned long cElements;
     long lLbound;
} SAFEARRAYBOUND;
```

SafeArrayAccessData Quick Info

HRESULT SafeArrayAccessData (

SAFEARRAY FAR* *psa,*
void HUGEP* FAR* *ppvdata*
);

Increments the lock count of an array, and retrieves a pointer to the array data.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

ppvdata

On exit, pointer to a pointer to the array data. Arrays may be larger than 64K, so very large pointers should be used only in Windows version 3.1 or later.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_UNEXPECTED	The array could not be locked.

Example

The following example sorts a safe array of one dimension that contains BSTRs by accessing the array elements directly. This approach is faster than using **SafeArrayGetElement** and **SafeArrayPutElement**.

```
long i, j, min;
BSTR bstrTemp;
BSTR HUGEP *pbstr;
HRESULT hr;

// Get a pointer to the the elements of the array.
hr = SafeArrayAccessData(psa, (void HUGEP* FAR*)&pbstr);
if (FAILED(hr))
goto error;

// Bubble sort.
cElements = lUBound-lLBound+1;
for (i = 0; i < cElements-1; i++)
{
     min = i;
     for (j = i+1; j < cElements; j++)
     {
          if (wcscmp(pbstr[j], pbstr[min]) < 0)
               min = j;
```

```
        }

        // Swap array[min] and array[i].
        bstrTemp = pbstr[min];
        pbstr[min] = pbstr[i];
        pbstr[i] = bstrTemp;

    }

    SafeArrayUnaccessData(psa);
```

SafeArrayAllocData Quick Info

HRESULT SafeArrayAllocData(

SAFEARRAY FAR* *psa*
);

Allocates memory for a safe array, based on a descriptor created with **SafeArrayAllocDescriptor**.

Parameter

psa

Pointer to an array descriptor created by **SafeArrayAllocDescriptor**.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_UNEXPECTED	The array could not be locked.

Example

The following example creates a safe array using the **SafeArrayAllocDescriptor** and **SafeArrayAllocData** functions.

```
SAFEARRAY FAR* FAR*ppsa;
unsigned int ndim =  2;
HRESULT hresult = SafeArrayAllocDescriptor(ndim, ppsa);
if( FAILED(hresult))
     return ERR_OutOfMemory;
(*ppsa)->rgsabound[ 0 ].lLbound = 0;
(*ppsa)->rgsabound[ 0 ].cElements = 5;
(*ppsa)->rgsabound[ 1 ].lLbound = 1;
(*ppsa)->rgsabound[ 1 ].cElements = 4;
hresult = SafeArrayAllocData(*ppsa);
if( FAILED(hresult)) {
     SafeArrayDestroyDescriptor(*ppsa)
     return ERR_OutOfMemory;
}
```

See Also

SafeArrayAllocData, **SafeArrayDestroyData**, **SafeArrayDestroyDescriptor**

SafeArrayAllocDescriptor Quick Info

HRESULT SafeArrayAllocDescriptor(

unsigned int *cDims,*
SAFEARRAY FAR* FAR* *ppsaOut*
);

Allocates memory for a safe array descriptor.

Parameters

cDims

The number of dimensions of the array.

ppsaOut

Pointer to a location in which to store the created array descriptor.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_UNEXPECTED	The array could not be locked.

Comments

This function allows the creation of safe arrays that contain elements with data types other than those provided by **SafeArrayCreate**. After creating an array descriptor using **SafeArrayAllocDescriptor**, set the element size in the array descriptor, an call **SafeArrayAllocData** to allocate memory for the array elements.

Example

The following example creates a safe array using the **SafeArrayAllocDescriptor** and **SafeArrayAllocData** functions.

```
SAFEARRAY FAR* FAR*ppsa;
unsigned int ndim =  2;
HRESULT hresult = SafeArrayAllocDescriptor( ndim, ppsa );
if( FAILED( hresult ) )
     return ERR_OutOfMemory;
(*ppsa)->rgsabound[ 0 ].lLbound = 0;
(*ppsa)->rgsabound[ 0 ].cElements = 5;
(*ppsa)->rgsabound[ 1 ].lLbound = 1;
(*ppsa)->rgsabound[ 1 ].cElements = 4;
hresult = SafeArrayAllocData( *ppsa );
if( FAILED( hresult ) ) {
     SafeArrayDestroyDescriptor( *ppsa )
     return ERR_OutOfMemory;
```

```
}
```

See Also

SafeArrayAllocData, SafeArrayDestroyData, SafeArrayDestroyDescriptor

SafeArrayCopy Quick Info

HRESULT SafeArrayCopy(

SAFEARRAY FAR* *psa,*
SAFEARRAY FAR* FAR* *ppsaOut*
);

Creates a copy of an existing safe array.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

ppsaOut

Pointer to a location in which to return the new array descriptor.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_OUTOFMEMORY	Insufficient memory to create the copy.

Comments

SafeArrayCopy calls the string or variant manipulation functions if the array to copy contains either of these data types. If the array being copied contains object references, the reference counts for the objects are incremented.

See Also

SysAllocStringLen, **VariantCopy**, **VariantCopyInd**

SafeArrayCopyData

HRESULT SafeArrayCopyData(

SAFEARRAY FAR* *psaSource,*
SAFEARRAY FAR* FAR* *ppsaTarget*
);

Copies the source array to the target array after releasing any resources in the target array. This is similar to **SafeArrayCopy,** except that the target array has to be set up by the caller. The target is not allocated or reallocated.

Parameters

psaSource

The safe array from which to be copied.

ppsaTarget

On exit, the array referred to by *ppsaTarget* contains a copy of the data in *psaSource*.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_OUTOFMEMORY	Insufficient memory to create the copy.

Comments

Visual Basic for Applications and Automation use the same set of rules with cases in which the size or types of source and destination arrays do not match. The rules of Visual Basic are described in the following comments.

Array Assignment

In general, VBA3 supports array assignment.

```
Dim lhs(1 To 10) As Integer
Dim rhs(1 To 10) As Integer

lhs = rhs
```

When the number of dimensions, the size of those dimensions, and the element types match, data types are differentiated based on the following factors:

- **Fixed-size, left side.** The left side is fixed if the type of the expression on the left side is a fixed-size array. For example, the following statement is a declaration of a fixed-size array.

  ```
  Dim x (1 To 10) As Integer
  ```

- **Matching number of dimensions.** The number of dimensions of the left side may or may not

match the number of dimensions of the array on the right side.

- **Dimensions match.** The dimensions match if, for each dimension, the number of elements match. The dimensions can match even if the declarations are slightly different, such as when one array is zero-based and another is one-based, but they have the same number of elements.

The following table shows what happens when the number of dimensions, size of the dimension, and element types do not match:

Fixed-size, left side	Number of dimensions	Dimensions match	What happens
No	Yes or No	Yes or No	Success. If necessary, the left side is resized to the size of the right side.
Yes	No		Failure.
Yes	Yes	No	Treated in same manner as fixed-length strings. If the right side has more elements than the left side, the assignment succeeds and the extra elements have no effect. If the left side has more elements than the right side, the assignment succeeds and the unaffected elements of the left side are zero-, null-, or empty-filled, depending on the types of the elements.
Yes	Yes	Yes	Success.

See Also

SysAllocStringLen, VariantCopy, VariantCopyInd

SafeArrayCreate Quick Info

HRESULT SafeArrayCreate(

VARTYPE *vt*,
unsigned int *cDims*,
SAFEARRRAYBOUND FAR* *rgsabound*
);

Creates a new array descriptor, allocates and initializes the data for the array, and returns a pointer to the new array descriptor.

Parameters

vt

The base type of the array (the VARTYPE of each element of the array). The VARTYPE is restricted to a subset of the variant types. Neither the VT_ARRAY nor the VT_BYREF flag can be set. VT_EMPTY and VT_NULL are not valid base types for the array. All other types are legal.

cDims

Number of dimensions in the array. The number cannot be changed after the array is created.

rgsabound

Pointer to a vector of bounds (one for each dimension) to allocate for the array.

Return Value

Points to the array descriptor, or Null if the array could not be created.

Example

```
HRESULT PASCAL __export CPoly::EnumPoints(IEnumVARIANT FAR* FAR* ppenum)
{
     unsigned int i;
     HRESULT hresult;
     VARIANT var;
     SAFEARRAY FAR* psa;
     CEnumPoint FAR* penum;
     POINTLINK FAR* ppointlink;
     SAFEARRAYBOUND rgsabound[1];
     rgsabound[0].lLbound = 0;
     rgsabound[0].cElements = m_cPoints;
     psa = SafeArrayCreate(VT_VARIANT, 1, rgsabound);
     if(psa == NULL){hresult = ReportResult(0, E_OUTOFMEMORY, 0, 0);
           goto LError0}

     // Code omitted here for brevity.

LError0:;
     return hresult;
}
```

SafeArrayCreateVector

HRESULT SafeArrayCreateVector(

VARTYPE *vt*,
long *lbound*,
unsigned int *cElements*
);

Creates a one-dimensional array whose lower bound is always zero. A safe array created with **SafeArrayCreateVector** is a fixed size, so the constant FADF_FIXEDSIZE is always set.

Parameters

vt

The base type of the array (the VARTYPE of each element of the array). The VARTYPE is restricted to a subset of the variant types. Neither the VT_ARRAY nor the VT_BYREF flag can be set. VT_EMPTY and VT_NULL are not valid base types for the array. All other types are legal.

lbound

The lower bound for the array. Can be negative.

cElements

The number of elements in the array.

Return Value

Points to the array descriptor, or Null if the array could not be created.

Comments

SafeArrayCreateVector allocates a single block of memory containing a SAFEARRAY structure for a single-dimension array (24 bytes), immediately followed by the array data. All of the existing safe array functions work correctly for safe arrays that are allocated with **SafeArrayCreateVector**.

A **SafeArrayCreateVector** is allocated as a single block of memory. Both the **SafeArray** descriptor and the array data block are allocated contiguously in one allocation, which speeds up array allocation. However, a user can allocate the descriptor and data area separately using the **SafeArrayAllocDescriptor** and **SafeArrayAllocData** calls.

SafeArrayDestroy Quick Info

HRESULT SafeArrayDestroy(

SAFEARRAY FAR* *psa*
);

Destroys an existing array descriptor and all of the data in the array. If objects are stored in the array, **Release** is called on each object in the array.

Parameter

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_ARRAYISLOCKED	The array is currently locked.
E_INVALIDARG	The item pointed to by *psa* is not a safe array descriptor.

Example

```
STDMETHODIMP_(ULONG) CEnumPoint::Release()
{
     if(--m_refs == 0){
          if(m_psa != NULL)
          SafeArrayDestroy(m_psa);
          delete this;
          return 0;
     }
     return m_refs;
}
```

SafeArrayDestroyData Quick Info

HRESULT SafeArrayDestroyData(

SAFEARRAY FAR* *psa*
);

Destroys all the data in a safe array.

Parameter

psa

Pointer to an array descriptor.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_ARRAYISLOCKED	The array is currently locked.
E_INVALIDARG	The item pointed to by *psa* is not a safe array descriptor.

Comments

This function is typically used when freeing safe arrays that contain elements with data types other than variants. If objects are stored in the array, **Release** is called on each object in the array.

See Also

SafeArrayAllocData, SafeArrayAllocDescriptor, SafeArrayDestroyDescriptor

SafeArrayDestroyDescriptor Quick Info

HRESULT SafeArrayDestroyDescriptor(

SAFEARRAY FAR* *psa*
);

Destroys a descriptor of a safe array.

Parameter

psa

Pointer to a safe array descriptor.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_ARRAYISLOCKED	The array is currently locked.
E_INVALIDARG	The item pointed to by *psa* is not a safe array descriptor.

Comments

This function is typically used to destroy the descriptor of a safe array that contains elements with data types other than variants. Destroying the array descriptor does not destroy the elements in the array. Before destroying the array descriptor, call **SafeArrayDestroyData** to free the elements.

See Also

SafeArrayAllocData, **SafeArrayAllocDescriptor**, **SafeArrayDestroyData**

SafeArrayGetDim Quick Info

HRESULT SafeArrayGetDim(

unsigned int **SafeArrayGetDim***(psa),*
SAFEARRAY FAR* *psa*
);

Returns the number of dimensions in the array.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

Return Value

Returns the number of dimensions in the array.

Example

```
HRESULT
CEnumPoint::Create(SAFEARRAY FAR* psa, CEnumPoint FAR* FAR* ppenum)
{
      long lBound;
      HRESULT hresult;
      CEnumPoint FAR* penum;

      // Verify that the SafeArray is the proper shape.
      if(SafeArrayGetDim(psa) != 1)
            return ReportResult(0, E_INVALIDARG, 0, 0);

      // Code omitted here for brevity.

}
```

SafeArrayGetElement Quick Info

HRESULT SafeArrayGetElement(

SAFEARRAY FAR* *psa*,
long FAR* *rgIndices*,
void FAR* *pvData*
);

Retrieves a single element of the array.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

rgIndices

Pointer to a vector of indexes for each dimension of the array. The right-most (least significant) dimension is *rgIndices*[0]. The left-most dimension is stored at *rgIndices*[*psa->cDims* - 1].

pvData

Pointer to the location to place the element of the array.

Comments

This function calls **SafeArrayLock** and **SafeArrayUnlock** automatically, before and after retrieving the element. The caller must provide a storage area of the correct size to receive the data. If the data element is a string, object, or variant, the function copies the element in the correct way.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_BADINDEX	The specified index is invalid.
E_INVALIDARG	One of the arguments is invalid.
E_OUTOFMEMORY	Memory could not be allocated for the element.

Example

```
STDMETHODIMP CEnumPoint::Next(
      ULONG celt,
      VARIANT FAR rgvar[],
      ULONG FAR* pceltFetched)
{
      unsigned int i;
      long ix;
      HRESULT hresult;

      for(i = 0; i < celt; ++i)
```

```
            VariantInit(&rgvar[i]);

      for(i = 0; i < celt; ++i){
            if(m_iCurrent == m_celts){
            hresult = ReportResult(0, S_FALSE, 0, 0);
                  goto LDone;
      }

            ix = m_iCurrent++;
            hresult = SafeArrayGetElement(m_psa, &ix, &rgvar[i]);
            if(FAILED(hresult))
                  goto LError0;
      }
      hresult = NOERROR;

LDone:;
      *pceltFetched = i;
      return hresult;

LError0:;
      for(i = 0; i < celt; ++i)
            VariantClear(&rgvar[i]);
      return hresult;
}
```

SafeArrayGetElemsize Quick Info

HRESULT SafeArrayGetElemsize(

unsigned int SafeArrayGetElemsize*(psa)*,
SAFEARRAY FAR* *psa*
);

Returns the size (in bytes) of the elements of a safe array.

Parameter

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

SafeArrayGetLBound Quick Info

HRESULT SafeArrayGetLBound(

SAFEARRAY FAR* *psa*,
unsigned int *nDim*,
long FAR* *plLbound*
);

Returns the lower bound for any dimension of a safe array.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

nDim

The array dimension for which to get the lower bound.

plLbound

Pointer to the location to return the lower bound.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_BADINDEX	The specified index is out of bounds.
E_INVALIDARG	One of the arguments is invalid.

Example

```
HRESULT
CEnumPoint::Create(SAFEARRAY FAR* psa, CEnumPoint FAR* FAR* ppenum)
{
      long lBound;
      HRESULT hresult;
      CEnumPoint FAR* penum;


      // Verify that the SafeArray is the proper shape.
      hresult = SafeArrayGetLBound(psa, 1, &lBound);
      if(FAILED(hresult))
            return hresult;

      // Code omitted here for brevity.


}
```

SafeArrayGetUBound Quick Info

HRESULT SafeArrayGetUBound(

SAFEARRAY FAR* *psa*,
unsigned int *nDim*,
long FAR* *plUbound*
);

Returns the upper bound for any dimension of a safe array.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate()**.

nDim

The array dimension for which to get the upper bound.

plUbound

Pointer to the location to return the upper bound.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_BADINDEX	The specified index is out of bounds.
E_INVALIDARG	One of the arguments is invalid.

Example

```
HRESULT
CEnumPoint::Create(SAFEARRAY FAR* psa, CEnumPoint FAR* FAR* ppenum)
{
     long lBound;
     HRESULT hresult;
     CEnumPoint FAR* penum;

     // Verify that the SafeArray is the proper shape.
     hresult = SafeArrayGetUBound(psa, 1, &lBound);
     if(FAILED(hresult))
           goto LError0;

     // Code omitted here for brevity.

LError0:;
     penum->Release();

     return hresult;
}
```

SafeArrayLock Quick Info

HRESULT SafeArrayLock(

SAFEARRAY FAR* *psa*
);

Increments the lock count of an array, and places a pointer to the array data in *pvData* of the array descriptor.

Parameter

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

Comments

The pointer in the array descriptor is valid until **SafeArrayUnlock** is called. Calls to **SafeArrayLock** can be nested. An equal number of calls to **SafeArrayUnlock** are required.

An array cannnot be deleted while it is locked.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_UNEXPECTED	The array could not be locked.

SafeArrayPtrOfIndex Quick Info

HRESULT SafeArrayPtrOfIndex(

SAFEARRAY FAR* *psa*,
long FAR* *rgIndices*,
void HUGEP* FAR* *ppvData*
);

Returns a pointer to an array element.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

rgIndices

An array of index values that identify an element of the array. All indexes for the element must be specified.

ppvData

On return, pointer to the element identified by the values in *rgIndices*.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
DISP_E_BADINDEX	The specified index was invalid.

Comments

The array should be locked before **SafeArrayPtrOfIndex** is called. Failing to lock the array can cause unpredictable results.

SafeArrayPutElement Quick Info

HRESULT SafeArrayPutElement(

SAFEARRAY FAR* *psa*,
long FAR* *rgIndices*,
void FAR* *pvData*
);

Assigns a single element to the array.

Parameters

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

rgIndices

Pointer to a vector of indexes for each dimension of the array. The right-most (least significant) dimension is *rgIndices*[0]. The left-most dimension is stored at *rgIndices*[*psa->cDims* - 1].

pvData

Pointer to the data to assign to the array. The variant types VT_DISPATCH, VT_UNKNOWN, and VT_BSTR are pointers, and do not require another level of indirection.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_BADINDEX	The specified index was invalid.
E_INVALIDARG	One of the arguments is invalid.
E_OUTOFMEMORY	Memory could not be allocated for the element.

Comments

This function automatically calls **SafeArrayLock** and **SafeArrayUnlock** before and after assigning the element. If the data element is a string, object, or variant, the function copies it correctly. If the existing element is a string, object, or variant, it is cleared correctly.

Note Multiple locks can be on an array. Elements can be put into an array while the array is locked by other operations.

Example

```
HRESULT PASCAL __export CPoly::EnumPoints(IEnumVARIANT FAR* FAR* ppenum)
{
     unsigned int i;
     HRESULT hresult;
```

```
    VARIANT var;
    SAFEARRAY FAR* psa;
    CEnumPoint FAR* penum;
    POINTLINK FAR* ppointlink;
    SAFEARRAYBOUND rgsabound[1];
    rgsabound[0].lLbound = 0;
    rgsabound[0].cElements = m_cPoints;

    psa = SafeArrayCreate(VT_VARIANT, 1, rgsabound);
    if(psa == NULL){
          hresult = ResultFromScode(E_OUTOFMEMORY);
          goto LError0;
    }

    // Code omitted here for brevity.

          V_VT(&var) = VT_DISPATCH;
          hresult = ppointlink->ppoint->QueryInterface(
          IID_IDispatch, (void FAR* FAR*)&V_DISPATCH(&var));
          if(hresult != NOERROR)
                goto LError1;

          ix[0] = i;
          SafeArrayPutElement(psa, ix, &var);

          ppointlink = ppointlink->next;
    }

    hresult = CEnumPoint::Create(psa, &penum);
    if(hresult != NOERROR)
          goto LError1;
    *ppenum = penum;
    return NOERROR;

LError1:;
    SafeArrayDestroy(psa);

LError0:;
    return hresult;
}
```

SafeArrayRedim Quick Info

HRESULT SafeArrayRedim(

SAFEARRAY FAR* *psa*,
SAFEARRAYBOUND FAR* *psaboundNew*
);

Changes the right-most (least significant) bound of a safe array.

Parameters

psa

Pointer to an array descriptor.

psaboundNew

Pointer to a new safe array bound structure that contains the new array boundary. You can change only the least significant dimension of an array.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_ARRAYISLOCKED	The array is currently locked.
E_INVALIDARG	The item pointed to by *psa* is not a safe array descriptor.

Comments

If you reduce the bound of an array, **SafeArrayRedim** deallocates the array elements outside the new array boundary. If the bound of an array is increased, **SafeArrayRedim** allocates and initializes the new array elements. The data is preserved for elements that exist in both the old and new array.

SafeArrayUnaccessData Quick Info

HRESULT SafeArrayUnaccessData(

SAFEARRAY FAR* *psa*
);

Decrements the lock count of an array, and invalidates the pointer retrieved by **SafeArrayAccessData**.

Parameter

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_UNEXPECTED	The array could not be unlocked.

SafeArrayUnlock Quick Info

HRESULT SafeArrayUnlock(

SAFEARRAY FAR* *psa*
);

Decrements the lock count of an array so it can be freed or resized.

Parameter

psa

Pointer to an array descriptor created by **SafeArrayCreate**.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_INVALIDARG	The argument *psa* was not a valid safe array descriptor.
E_UNEXPECTED	The array could not be unlocked.

Comments

This function is called after access to the data in an array is finished.

String Manipulation Functions

To handle strings that are allocated by one component and freed by another, Automation defines a special set of functions. These functions use the following data type:

```
typedef OLECHAR FAR* BSTR;
```

These strings are zero-terminated, and in most cases they can be treated just like OLECHAR* strings. However, you can query a BSTR (Basic string) for its length rather than scan it, so it can contain embedded null characters. The length is stored as an integer at the memory location preceding the data in the string. Instead of reading this location directly, applications should use the string manipulation functions to access the length of a BSTR.

In situations where a BSTR will not be translated from ANSI to Unicode, or vice versa, you can use BSTRs to pass binary data. For example, if code will run only on 16-bit systems and interact only with other 16-bit systems, you can use BSTRs. The preferred method of passing binary data is to use a SAFEARRAY of VT_UI1.

In 32-bit OLE, BSTRs use Unicode like all other strings in 32-bit OLE. In 16-bit OLE, BSTRs use ANSI. Win32 provides **MultiByteToWideChar** and **WideCharToMultiByte** to convert ANSI strings to Unicode, and Unicode strings to ANSI. Automation caches the space allocated for BSTRs. This speeds up the **SysAllocString**/**SysFreeString** sequence. However, this may also cause **IMallocSpy** to assign leaks to the wrong memory user because it is not aware of the caching done by Automation.

For example, if the application allocates a BSTR and frees it, the free block of memory is put into the BSTR cache by Automation. If the application then allocates another BSTR, it can get the free block from the cache. If the second BSTR allocation is not freed, **IMallocSpy** will attribute the leak to the first allocation of the BSTR. You can determine the correct source of the leak (the second allocation) by disabling the BSTR caching using the debug version of Oleaut32.dll, and by setting the environment variable OANOCACHE=1 before running the application.

A null pointer is a valid value for a BSTR variable. By convention, it is always treated the same as a pointer to a BSTR that contains zero characters. Also by convention, calls to functions that take a BSTR reference parameter must pass either a null pointer, or a pointer to an allocated BSTR. If the implementation of a function that takes a BSTR reference parameter assigns a new BSTR to the parameter, it must free the previously referenced BSTR.

SysAllocString Quick Info

BSTR SysAllocString(

OLECHAR FAR* *sz*
);

Allocates a new string and copies the passed string into it. Returns Null if there is insufficient memory, and if Null, Null is passed in.

Parameter

sz

A zero-terminated string to copy. The *sz* parameter must be a Unicode string in 32-bit applications, and an ANSI string in 16-bit applications.

Return Value

If successful, points to a BSTR containing the string. If insufficient memory exists or *sz* was Null, returns Null.

Comments

You can free strings created with **SysAllocString** using **SysFreeString**.

Example

```
inline void CStatBar::SetText(OLECHAR FAR* sz)
{
     SysFreeString(m_bstrMsg);
     m_bstrMsg = SysAllocString(sz);
}
```

SysAllocStringByteLen Quick Info

BSTR SysAllocStringByteLen(

char FAR* *psz*,
unsigned int *len*
);

Takes an ANSI string as input, and returns a BSTR that contains an ANSI string. Does not perform any ANSI-to-Unicode translation.

Parameters

psz

A zero-terminated string to copy, or Null to keep the string uninitialized.

len

Number of bytes to copy from *psz*. A null character is placed afterwards, allocating a total of *len*+1 bytes.

Allocates a new string of *len* bytes, copies *len* bytes from the passed string into it, and then appends a null character. Valid only for 32-bit systems.

Return Value

Points to a copy of the string, or Null if insufficient memory exists.

Comments

This function is provided to create BSTRs that contain binary data. You can use this type of BSTR only in situations where it will not be translated from ANSI to Unicode, or vice versa.

For example, do not use these BSTRs between a 16-bit and a 32-bit application running on a 32-bit Windows system. The OLE 16-bit to 32-bit (and 32-bit to 16-bit) interoperability layer will translate the BSTR and corrupt the binary data. The preferred method of passing binary data is to use a SAFEARRAY of VT_UI1, which will not be translated by OLE.

If *psz* is Null, a string of the requested length is allocated, but not initialized. The string *psz* can contain embedded null characters, and does not need to end with a Null. Free the returned string later with **SysFreeString**.

SysAllocStringLen Quick Info

BSTR SysAllocStringLen(

OLECHAR FAR* *pch*,
unsigned int *cch*
);

Allocates a new string, copies *cch* characters from the passed string into it, and then appends a null character.

Parameters

pch

A pointer to *cch* characters to copy, or Null to keep the string uninitialized.

cch

Number of characters to copy from *pch.* A null character is placed afterwards, allocating a total of *cch*+1 characters.

Return Value

Points to a copy of the string, or Null if insufficient memory exists.

Comments

If *pch* is Null, a string of the requested length is allocated, but not initialized. The *pch* string can contain embedded null characters and does not need to end with a Null. Free the returned string later with **SysFreeString**.

SysFreeString Quick Info

void SysFreeString(

BSTR *bstr*
);

Frees a string allocated previously by **SysAllocString**, **SysAllocStringByteLen**, **SysReAllocString**, **SysAllocStringLen**, or **SysReAllocStringLen**.

Parameter

bstr

A BSTR allocated previously, or Null. If Null, the function simply returns.

Return Value

None.

Example

```
CStatBar::~CStatBar()
{
     SysFreeString(m_bstrMsg);
}
```

SysReAllocString Quick Info

BOOL SysReAllocString(

BSTR FAR* *pbstr*,
OLECHAR FAR* *sz*
);

Allocates a new BSTR and copies the passed string into it, then frees the BSTR referenced by *pbstr,* and finally resets *pbstr* to point to the new BSTR.

Parameters

pbstr

Points to a variable containing a BSTR.

sz

A zero-terminated string to copy.

Return Value

Returns False if insufficient memory exists.

SysReAllocStringLen Quick Info

BOOL SysReAllocStringLen(

BSTR FAR* *pbstr*,
OLECHAR FAR* *pch*,
unsigned int *cch*
);

Creates a new BSTR containing a specified number of characters from an old BSTR, and frees the old BSTR.

Parameters

pbstr

Pointer to a variable containing a BSTR.

pch

Pointer to *cch* characters to copy, or Null to keep the string uninitialized.

cch

Number of characters to copy from *pch.* A null character is placed afterward, allocating a total of *cch*+1 characters.

Return Value

Returns True if the string is reallocated successfully, or False if insufficient memory exists.

Comments

Allocates a new string, copies *cch* characters from the passed string into it, and then appends a null character. Frees the BSTR referenced currently by *pbstr*, and resets *pbstr* to point to the new BSTR. If *pch* is Null, a string of length *cch* is allocated but not initialized.

The *pch* string can contain embedded null characters and does not need to end with a Null.

SysStringByteLen Quick Info

unsigned int SysStringByteLen(

BSTR *bstr*
);

Returns the length (in bytes) of a BSTR. Valid for 32-bit systems only.

Parameter

bstr

A BSTR allocated previously. It cannot be Null.

Return Value

The number of bytes in *bstr*, not including a terminating null character.

Comments

The returned value may be different from **fstrlen(***bstr***)** if the BSTR was allocated with **Sys[Re]AllocStringLen** or **SysAllocStringByteLen**, and the passed-in characters included a null character in the first *len* characters. For a BSTR allocated with **Sys[Re]AllocStringLen** or **SysAllocStringByteLen**, this function always returns the number of bytes specified in the *len* parameter at allocation time.

Example

```
// Display the status message.

TextOut(
     hdc,
     rcMsg.left + (m_dxFont / 2),
     rcMsg.top + ((rcMsg.bottom - rcMsg.top - m_dyFont) / 2),
     m_bstrMsg, SysStringByteLen(m_bstrMsg));
```

SysStringLen Quick Info

unsigned int SysStringLen(

BSTR *bstr*
);

Returns the length of a BSTR.

Parameter

bstr

A BSTR allocated previously. Cannot be Null.

Return Value

The number of characters in *bstr*, not including a terminating null character.

Comments

The returned value may be different from **_fstrlen(***bstr***)** if the BSTR was allocated with **Sys[Re]AllocStringLen** or **SysAllocStringByteLen**, and the passed-in characters included a null character in the first *cch* characters. For a BSTR allocated with **Sys[Re]AllocStringLen** or **SysAllocStringByteLen**, this function always returns the number of characters specified in the *cch* parameter at allocation time.

Example

```
// Display the status message.
//
TextOut(
     hdc,
     rcMsg.left + (m_dxFont / 2),
     rcMsg.top + ((rcMsg.bottom - rcMsg.top - m_dyFont) / 2),
     m_bstrMsg, SysStringLen(m_bstrMsg));
```

Variant Manipulation API Functions

These functions are provided to allow applications to manipulate VARIANTARG variables. Applications that implement **IDispatch** should test each VARIANTARG for all permitted types by attempting to coerce the variant to each type using **VariantChangeType** or **VariantChangeTypeEx**. If objects are allowed, the application should always test for object types before other types. If an object type is expected, the application must use **IUnknown::QueryInterface** to test whether the object is the desired type.

Although applications can access and interpret the VARIANTARGs without these functions, using them ensures uniform conversion and coercion rules for all implementors of **IDispatch**. For example, these functions automatically coerce numeric arguments to strings, and vice versa, when necessary.

Because variants can contain strings, references to scalars, objects, and arrays, all data ownership rules must be followed. All variant manipulation functions should conform to the following rules:

1. Before use, all VARIANTARGs must be initialized by **VariantInit**.
2. For the types VT_UI1, VT_I2, VT_I4, VT_R4, VT_R8, VT_BOOL, VT_ERROR, VT_CY, and VT_DATE, data is stored within the VARIANT structure. Any pointers to the data become invalid when the type of the variant is changed.
3. For VT_BYREF | any type, the memory pointed to by the variant is owned and freed by the caller of the function.
4. For VT_BSTR, there is only one owner for the string. All strings in variants must be allocated with the **SysAllocString** function. When releasing or changing the type of a variant with the VT_BSTR type, **SysFreeString** is called on the contained string.
5. For VT_ARRAY | any type, the rule is analogous to the rule for VT_BSTR. All arrays in variants must be allocated with **SafeArrayCreate**. When releasing or changing the type of a variant with the VT_ARRAY flag set, **SafeArrayDestroy** is called.
6. For VT_DISPATCH and VT_UNKNOWN, the objects that are pointed to have reference counts that are incremented when they are placed in a variant. When releasing or changing the type of the variant, **Release** is called on the object that is pointed to.

VariantChangeType Quick Info

HRESULT VariantChangeType(

VARIANTARG FAR* *pvargDest*,
VARIANTARG FAR* *pvargSrc*,
unsigned short *wFlags*,
VARTYPE *vtNew*
);

Converts a variant from one type to another.

Parameters

pvargDest

Pointer to the VARIANTARG to receive the coerced type. If this is the same as *pvargSrc*, the variant will be converted in place.

pvargSrc

Pointer to the source VARIANTARG to be coerced.

wFlags

Flags that control the coercion. The only defined flag is VARIANT_NOVALUEPROP, which prevents the function from attempting to coerce an object to a fundamental type by getting the **Value** property. Applications should set this flag only if necessary, because it makes their behavior inconsistent with other applications.

vtNew

The type to coerce to. If the return code is S_OK, the *vt* field of the **pvargDest* is always the same as this value.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_BADVARTYPE	The variant type *vtNew* is not a valid type of variant.
DISP_E_OVERFLOW	The data pointed to by *pvargSrc* does not fit in the destination type.
DISP_E_TYPEMISMATCH	The argument could not be coerced to the specified type.
E_INVALIDARG	One of the arguments is invalid.
E_OUTOFMEMORY	Memory could not be allocated for the conversion.

Comments

The **VariantChangeType** function handles coercions between the fundamental types (including numeric-to-string and string-to-numeric coercions). A variant that has VT_BYREF set is coerced to a value by obtaining the referenced value. An object is coerced to a value by invoking the object's **Value** property

(DISPID_VALUE).

Typically, the implementor of **IDispatch::Invoke** determines which member is being accessed, and then calls **VariantChangeType** to get the value of one or more arguments. For example, if the **IDispatch** call specifies a **SetTitle** member that takes one string argument, the implementor would call **VariantChangeType** to attempt to coerce the argument to VT_BSTR. If **VariantChangeType** does not return an error, the argument could then be obtained directly from the *bstrVal* field of the VARIANTARG. If **VariantChangeType** returns DISP_E_TYPEMISMATCH, the implementor would set **puArgErr* to 0 (indicating the argument in error) and return DISP_E_TYPEMISMATCH from **IDispatch::Invoke**.

Arrays of one type cannnot be converted to arrays of another type with this function.

Note The type of a VARIANTARG should not be changed in the *rgvarg* array in place.

See Also

VariantChangeTypeEx

VariantChangeTypeEx Quick Info

HRESULT VariantChangeTypeEx(

VARIANTARG FAR* *pvargDest*,
VARIANTARG FAR* *pvargSrc*,
LCID *lcid*,
unsigned short *wFlags*,
VARTYPE *vtNew*
);

Converts a variant from one type to another, using a locale ID.

Parameters

pvargDest

Pointer to the VARIANTARG to receive the coerced type. If this is the same as *pvargSrc*, the variant will be converted in place.

pvargSrc

Pointer to the source VARIANTARG to be coerced.

lcid

The locale ID for the variant to coerce. The locale ID is useful when the type of the source or destination VARIANTARG is VT_BSTR, VT_DISPATCH, or VT_DATE.

wFlags

Flags that control the coercion. The only defined flag is VARIANT_NOVALUEPROP, which prevents the function from attempting to coerce an object to a fundamental type by getting its **Value** property. Applications should set this flag only if necessary, because it makes their behavior inconsistent with other applications.

vtNew

The type to coerce to. If the return code is S_OK, the *vt* field of the **pvargDest* is guaranteed to be equal to this value.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_BADVARTYPE	The variant type *vtNew* is not a valid type of variant.
DISP_E_OVERFLOW	The data pointed to by *pvargSrc* does not fit in the destination type.
DISP_E_TYPEMISMATCH	The argument could not be coerced to the specified type.
E_INVALIDARG	One of the arguments is invalid.
E_OUTOFMEMORY	Memory could not be allocated for the conversion.

Comments

The **VariantChangeTypeEx** function handles coercions between the fundamental types (including numeric-to-string and string-to-numeric coercions). To change a type with the VT_BYREF flag set to one without VT_BYREF, change the referenced value to **VariantChangeTypeEx**. To coerce objects to fundamental types, obtain the value of the **Value** property.

Typically, the implementor of **IDispatch::Invoke** determines which member is being accessed, and then calls **VariantChangeType** to get the value of one or more arguments. For example, if the **IDispatch** call specifies a **SetTitle** member that takes one string argument, the implementor would call **VariantChangeTypeEx** to attempt to coerce the argument to VT_BSTR.

If **VariantChangeTypeEx** does not return an error, the argument could then be obtained directly from the *bstrVal* field of the VARIANTARG. If **VariantChangeTypeEx** returns DISP_E_TYPEMISMATCH, the implementor would set **puArgErr* to 0 (indicating the argument in error) and return DISP_E_TYPEMISMATCH from **IDispatch::Invoke**.

Arrays of one type cannot be converted to arrays of another type with this function.

Note The type of a VARIANTARG should not be changed in the *rgvarg* array in place.

See Also

VariantChangeType

VariantClear Quick Info

HRESULT VariantClear(

VARIANTARG FAR* *pvarg*
);

Clears a variant.

Parameter

pvarg

Pointer to the VARIANTARG to clear.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_ARRAYISLOCKED	The variant contains an array that is locked.
DISP_E_BADVARTYPE	The variant type *pvarg* is not a valid type of variant.
E_INVALIDARG	One of the arguments is invalid.

Comments

Use this function to clear variables of type VARIANTARG (or VARIANT) before the memory containing the VARIANTARG is freed (as when a local variable goes out of scope).

The function clears a VARIANTARG by setting the *vt* field to VT_EMPTY and the *wReserved* field to 0. The current contents of the VARIANTARG are released first. If the *vt* field is VT_BSTR, the string is freed. If the *vt* field is VT_DISPATCH, the object is released. If the *vt* field has the VT_ARRAY bit set, the array is freed.

In certain cases, it may be preferable to clear a variant in code without calling **VariantClear**. For example, you can change the type of a VT_I4 variant to another type without calling this function. However, you must call **VariantClear** if a VT_type is received but cannot be handled. Using **VariantClear** in these cases ensures that code will continue to work if Automation adds new variant types in the future.

Example

```
for(i = 0; i < celt; ++i)
     VariantClear(&rgvar[i]);
```

VariantCopy Quick Info

HRESULT VariantCopy(

VARIANTARG FAR* *pvargDest*,
VARIANTARG FAR* *pvargSrc*
);

Frees the destination variant and makes a copy of the source variant.

Parameters

pvargDest

Pointer to the VARIANTARG to receive the copy.

pvargSrc

Pointer to the VARIANTARG to be copied.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_ARRAYISLOCKED	The variant contains an array that is locked.
DISP_E_BADVARTYPE	The source and destination have an invalid variant type (usually uninitialized).
E_OUTOFMEMORY	Memory could not be allocated for the copy.
E_INVALIDARG	The argument *pvargSrc* was VT_BYREF.

Comments

First, free any memory that is owned by *pvargDest*, such as **VariantClear** (*pvargDest* must point to a valid initialized variant, and not simply to an uninitialized memory location). Then *pvargDest* receives an exact copy of the contents of *pvargSrc*.

If *pvargSrc* is a VT_BSTR, a copy of the string is made. If *pvargSrc* is a VT_ARRAY, the entire array is copied. If *pvargSrc* is a VT_DISPATCH or VT_UNKNOWN, **AddRef** is called to increment the object's reference count.

VariantCopyInd Quick Info

HRESULT VariantCopyInd(

VARIANT FAR* *pvarDest*,
VARIANTARG FAR* *pvargSrc*
);

Frees the destination variant and makes a copy of the source VARIANTARG, performing the necessary indirection if the source is specified to be VT_BYREF.

Parameters

pvarDest

Pointer to the VARIANTARG that will receive the copy.

pvargSrc

Pointer to the VARIANTARG that will be copied.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_ARRAYISLOCKED	The variant contains an array that is locked.
DISP_E_BADVARTYPE	The source and destination have an invalid variant type (usually uninitialized).
E_OUTOFMEMORY	Memory could not be allocated for the copy.
E_INVALIDARG	The argument *pvargSrc* was VT_ARRAY.

Comments

This function is useful when a copy of a variant is needed, and to guarantee that it is not VT_BYREF, such as when handling arguments in an implementation of **IDispatch::Invoke**.

For example, if the source is a (VT_BYREF | VT_I2), the destination will be a BYVAL | VT_I2. The same is true for all legal VT_BYREF combinations, including VT_VARIANT.

If *pvargSrc* is (VT_BYREF | VT_VARIANT), and the contained variant is VT_BYREF, the contained variant is also dereferenced.

This function frees any existing contents of *pvarDest*.

VariantInit Quick Info

void VariantInit(

VARIANTARG FAR* *pvarg*
);

Initializes a variant.

Parameter

pvarg

Pointer to the VARIANTARG that will be initialized.

Comments

The **VariantInit** function initializes the VARIANTARG by setting the *vt* field to VT_EMPTY. Unlike **VariantClear**, this function does not interpret the current contents of the VARIANTARG. Use **VariantInit** to initialize new local variables of type VARIANTARG (or VARIANT).

Example

```
for(i = 0; i < celt; ++i)
     VariantInit(&rgvar[i]);
```

Data Type Conversion APIs

The files Oleaut32.dll (for 32-bit systems) and Ole2disp.dll (for 16-bit systems) provide the following low-level functions for converting variant data types. Higher-level variant manipulation functions (such as **VariantChangeType**) use these functions, but they can also be called directly.

Convert to type	From type	Function
unsigned char	**unsigned char**	None
	short	**VarUI1FromI2(***sIn, pbOut***)**
	long	**VarUI1FromI4(***lIn*, *pbOut***)**
	float	**VarUI1FromR4(***fltIn*, *pbOut***)**
	double	**VarUI1FromR8(***dblIn*, *pbOut***)**
	CURRENCY	**VarUI1FromCy(***cyIn*, *pbOut***)**
	DATE	**VarUI1FromDate(***dateIn*, *pbOut***)**
	OLECHAR FAR*	**VarUI1FromStr(***strIn*, *lcid*, *dwFlags, pbOut***)**
	IDispatch FAR*	**VarUI1FromDisp(***pdispIn*, *lcid, dwFlags, pbOut***)**
	BOOL	**VarUI1FromBool(***boolIn*, *pbOut***)**
short	**unsigned char**	**VarI2FromUI1(***bIn, psOut***)**
	short	None
	long	**VarI2FromI4(***lIn*, *psOut***)**
	float	**VarI2FromR4(***fltIn*, *psOut***)**
	double	**VarI2FromR8(***dblIn*, *psOut***)**
	CURRENCY	**VarI2FromCy(***cyIn*, *psOut***)**
	DATE	**VarI2FromDate(***dateIn*, *psOut***)**
	OLECHAR FAR*	**VarI2FromStr(***strIn*, *lcid*, *dwFlags, psOut***)**
	IDispatch FAR*	**VarI2FromDisp(***pdispIn*, *lcid, dwFlags, psOut***)**
	BOOL	**VarI2FromBool(***boolIn*, *psOut***)**
long	**unsigned char**	**VarI4FromUI1(***bIn*, *plOut***)**
	short	**VarI4FromI2(***sIn*, *plOut***)**
	long	None
	float	**VarI4FromR4(***fltIn*, *plOut***)**
	double	**VarI4FromR8(***dblIn*, *plOut***)**
	CURRENCY	**VarI4FromCy(***cyIn*, *plOut***)**
	DATE	**VarI4FromDate(***dateIn*, *plOut***)**
	OLECHAR FAR*	**VarI4FromStr(***strIn*, *lcid*, *dwFlags, plOut***)**
	IDispatch FAR*	**VarI4FromDisp(***pdispIn*, *lcid*, *dwFlags, plOut***)**
	BOOL	**VarI4FromBool(***boolIn*,

		plOut)
float	**unsigned char**	**VarR4FromUI1(***bIn*, *prOut***)**
	short	**VarR4FromI2(***sIn*, *prOut***)**
	long	**VarR4FromI4**(*lIn*, *prOut*)
	float	None
	double	**VarR4FromR8(***dblIn*, *prOut***)**
	CURRENCY	**VarR4FromCy(***cyIn*, *prOut***)**
	DATE	**VarR4FromDate(***dateIn*, *prOut***)**
	OLECHAR FAR*	**VarR4FromStr(***strIn*, *lcid*, *dwFlags, prOut***)**
	IDispatch FAR*	**VarR4FromDisp(***pdispIn*, *lcid*, *dwFlags, prOut***)**
	BOOL	**VarR4FromBool(***boolIn*, *prOut***)**
Convert to type	**From type**	**Function**
double	**unsigned char**	**VarR8FromUI1(***bIn*, *pdblOut***)**
	short	**VarR8FromI2(***sIn*, *pdblOut***)**
	long	**VarR8FromI4(***lIn*, *pdblOut***)**
	float	**VarR8FromR4(***fltIn*, *pdblOut***)**
	double	None
	CURRENCY	**VarR8FromCy**(*cyIn*, *pdblOut***)**
	DATE	**VarR8FromDate(***dateIn*, *pdblOut***)**
	OLECHAR FAR*	**VarR8FromStr(***strIn*, *lcid*, *dwFlags, pdblOut***)**
	IDispatch FAR*	**VarR8FromDisp(***pdispIn*, *lcid*, *dwFlags, pdblOut***)**
	BOOL	**VarR8FromBool(***boolIn*, *pdblOut***)**
DATE	**unsigned char**	**VarDateFromUI1(***bIn*, *pdateOut***)**
	short	**VarDateFromI2(***sIn*, *pdateOut***)**
	long	**VarDateFromI4(***lIn*, *pdateOut***)**
	float	**VarDateFromR4(***fltIn*, *pdateOut***)**
	double	**VarDateFromR8(***dblIn*, *pdateOut***)**
	CURRENCY	**VarDateFromCy(***cyIn*, *pdateOut***)**
	DATE	None
	OLECHAR FAR*	**VarDateFromStr(***strIn*, *lcid*, *dwFlags, pdateOut***)**
	IDispatch FAR*	**VarDateFromDisp(***pdispIn*, *lcid*, *dwFlags, pdateOut***)**
	BOOL	**VarDateFromBool(***boolIn*,

		pdateOut)
CURRENCY	**unsigned char**	**VarCyFromUI1(***bIn*, *pcyOut***)**
	short	**VarCyFromI2(***sIn*, *pcyOut***)**
	long	**VarCyFromI4(***lIn*, *pcyOut***)**
	float	**VarCyFromR4(***fltIn*, *pcyOut***)**
	double	**VarCyFromR8(***dblIn*, *pcyOut***)**
	CURRENCY	None
	DATE	**VarCyFromDate(***dateIn, pcyOut***)**
	OLECHAR FAR*	**VarCyFromStr(***strIn*, *lcid*, *dwFlags, pcyOut***)**
	IDispatch FAR*	**VarCyFromDisp(***pdispIn*, *lcid*, *dwFlags, pcyOut***)**
	BOOL	**VarCyFromBool(***boolIn*, *pcyOut***)**
BSTR	**unsigned char**	**VarBstrFromUI1(***bIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	short	**VarBstrFromI2(***sIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	long	**VarBstrFromI4(***lIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	float	**VarBstrFromR4(***fltIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	double	**VarBstrFromR8(***dblIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	CURRENCY	**VarBstrFromCy(***cyIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	DATE	**VarBstrFromDate(***dateIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	OLECHAR FAR*	None
	IDispatch FAR*	**VarBstrFromDisp(***pdispIn*, *lcid*, *dwFlags*, *pbstrOut***)**
	BOOL	**VarBstrFromBool**(*boolIn*, *lcid*, *dwFlags*, *pbstrOut***)**
BOOL	**unsigned char**	**VarBoolFromUI1(***bIn*, *pboolOut***)**
	short	**VarBoolFromI2(***sIn*, *pboolOut***)**
	long	**VarBoolFromI4(***lIn*, *pboolOut***)**
	float	**VarBoolFromR4(***fltIn*, *pboolOut***)**
	double	**VarBoolFromR8(***dblIn*, *pboolOut***)**
	CURRENCY	**VarBoolFromCy(***cyIn*, *pboolOut***)**
	DATE	**VarBoolFromDate(***dateIn*, *pboolOut***)**
	OLECHAR FAR*	**VarBoolFromStr(***strIn*, *lcid*,

	dwFlags, pboolOut)
IDispatch FAR*	**VarBoolFromDisp(***pdispIn*, *lcid*, *dwFlags, pboolOut***)**
BOOL	None

Parameters

bIn, sIn, lIn, fltIn, dblIn, cyIn, dateIn, strIn, pdispIn, boolIn

The value to coerce. These parameters have the following data types:

Parameter	Data type
bIn	**unsigned char**
sIn	**short**
lIn	**long**
fltIn	**float**
dblIn	**double**
cyIn	**CURRENCY**
dateIn	**DATE**
strIn	**OLECHAR FAR***
pdispIn	**IDispatch FAR***
boolIn	**BOOL**

lcid

For conversions from string and VT_DISPATCH input, the locale ID to use for the conversion. For a list of locale IDs, see "Supporting Multiple National Languages" in Chapter 2, "Exposing Automation Objects."

dwFlags

One or more of the following flags:

Flag	Description
LOCALE_NOUSEROVERRIDE	Uses the system default locale settings, rather than custom locale settings.
VAR_TIMEVALUEONLY	Omits the date portion of a VT_DATE and returns only the time. Applies to conversions to or from dates.
VAR_DATEVALUEONLY	Omits the time portion of a VT_DATE and returns only the time. Applies to conversions to or from dates.

pbOut, psOut, plOut, pfltOut, pdblOut, pcyOut, pstrOut, pdispOut, pboolOut

A pointer to the coerced value. These parameters have the following data types:

Parameter	Data type
pbOut	**unsigned char**
psOut	**short**
plOut	**long**
pfltOut	**float**
pdblOut	**double**
pcyOut	**CURRENCY**
pdateOut	**DATE**
pstrOut	**OLECHAR FAR***
pdispOut	**IDispatch FAR***
pboolOut	**BOOL**

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
DISP_E_BADVARTYPE	The input parameter is not a valid type of variant.
DISP_E_OVERFLOW	The data pointed to by the output parameter does not fit in the destination type.
DISP_E_TYPEMISMATCH	The argument could not be coerced to the specified type.
E_INVALIDARG	One of the arguments is invalid.
E_OUTOFMEMORY	Memory could not be allocated for the conversion.

BSTR and Vector Conversion Functions

Automation supports conversion between an array of bytes and a BSTR through the two low-level conversion functions **VectorFromBstr** and **BstrFromVector**, and by performing the appropriate conversions in **VariantChangeType**, **ITypeInfo::Invoke**, **DispInvoke**, and other relevant locations.

BSTRs are wide, double-byte (Unicode) strings on 32-bit Windows platforms, and narrow, single-byte strings on the Apple PowerMac. These functions do not perform any special string handling. They simply move bytes from one location to another, so the width of strings does not affect these API functions.

VectorFromBstr

HRESULT VectorFromBstr(

BSTR *bstr*,
SAFEARRAY FAR* FAR* *ppsa*
);

Returns a vector, assigning each character in the BSTR to an element of the vector.

Parameters

bstr

The BSTR to be converted to a vector.

ppsa

On exit, *ppsa* points to a one-dimensional safe array containing the characters in the BSTR.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	BSTR is Null.

BstrFromVector

HRESULT BstrFromVector(

SAFEARRAY FAR* *psa*,
BSTR FAR* *pbstr*
);

Returns a BSTR, assigning each element of the vector to a character in the BSTR.

Parameters

psa

The vector to be converted to a BSTR.

pbstr

On exit, *pbstr* points to a BSTR, each character of which is assigned to an element from the vector.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	The argument *psa* is Null.
DISP_E_TYPEMISMATCH	The argument *psa* is not a vector (not an array of bytes).

Numeric Parsing Functions

Internally, Automation uses a single, fast-parsing function for all conversions of a string to a numeric type. Parsing a string to a number is a very common operation, so exposing this internal function will save duplication of code and provide consistent results and performance. For example, Visual Basic for Applications uses this function to implement file input of the DECIMAL data type.

```
// Flags used by both dwInFlags and dwOutFlags:
#define NUMPRS_LEADING_WHITE 0x0001
#define NUMPRS_TRAILING_WHITE0x0002
#define NUMPRS_LEADING_PLUS  0x0004
#define NUMPRS_TRAILING_PLUS 0x0008
#define NUMPRS_LEADING_MINUS 0x0010
#define NUMPRS_TRAILING_MINUS0x0020
#define NUMPRS_HEX_OCT       0x0040
#define NUMPRS_PARENS        0x0080
#define NUMPRS_DECIMAL       0x0100
#define NUMPRS_THOUSANDS           0x0200
#define NUMPRS_CURRENCY      0x0400
#define NUMPRS_EXPONENT      0x0800
#define NUMPRS_USE_ALL       0x1000
#define NUMPRS_STD           0x1FFF

// Flags used by dwOutFlags only:
#define NUMPRS_NEG           0x10000
#define NUMPRS_INEXACT       0x20000

// VarNumFromParseNum flags that indicate acceptable result types:
#define VTBIT_I1                (1 << VT_I1)
#define VTBIT_UI1               (1 << VT_UI1)
#define VTBIT_I2                (1 << VT_I2)
#define VTBIT_UI2               (1 << VT_UI2)
#define VTBIT_I4                (1 << VT_I4)
#define VTBIT_UI4               (1 << VT_UI4)
#define VTBIT_R4                (1 << VT_R4)
#define VTBIT_R8                (1 << VT_R8)
#define VTBIT_CY                (1 << VT_CY)
#define VTBIT_DECIMAL           (1 << VT_DECIMAL)
```

VarParseNumFromStr

HRESULT VarParseNumFromStr(

[in] OLECHAR* *strIn*,
[in] LCID *lcid*,
[in] unsigned long *dwFlags*,
[in] NUMPARSE **pnumprs*,
[out] unsigned char **rgbDig*
);

Parses a string, and creates a type-independent description of the number it represents. The first three parameters are identical to the first three parameters of **VarI2FromStr, VarI4FromStr, VarR8FromStr,** and so on. The fourth parameter is a pointer to a NUMPARSE structure, which contains both input information to the function as well as the results, as described above. The last parameter is a pointer to an array of digits, filled in by the function.

The **VarParseNumFromStr** function fills in the *dwOutFlags* element with each corresponding feature that was actually found in the string. This allows the caller to make decisions about what numeric type to use for the number, based on the format in which it was entered. For example, one application might want to use the CURRENCY data type if the currency symbol is used, and others may want to force a floating point type if an exponent was used.

Parameters

[in] *strIn*

Input string to be converted to a number.

lcid

Locale identifier

dwFlags

Allows the caller to control parsing, therefore defining the acceptable syntax of a number. If this field is set to zero, the input string must contain nothing but decimal digits. Setting each defined flag bit enables parsing of that syntactic feature. Standard Automation parsing (for example, as used by **VarI2FromStr**) has all flags set (NUMPRS_STD).

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Internal memory allocation failed. (Used for DBCS only to create a copy with all wide characters mapped narrow.)
DISP_E_TYPEMISMATCH	There is no valid number in the string, or there is no closing parenthesis to match an opening one. In the former case, *cDig* and *cchUsed* in the NUMPARSE structure will be zero. In the latter,

	the NUMPARSE structure and digit array are fully updated, as if the closing parenthesis was present.
DISP_E_OVERFLOW	For hexadecimal and octal digists, there are more digits than will fit into the array. For decimal, the exponent exceeds the maximum possible. In both cases, the NUMPARSE structure and digit array are fully updated (for decimal, the *cchUsed* field excludes the entire exponent).

NUMPARSE Structure

The caller of **VarParseNumFromStr()** must initialize two elements of the passed-in NUMPARSE structure:

```
typedef struct {
     int   cDig;
     unsigned long     dwInFlags;
     unsigned long     dwOutFlags;
     int   cchUsed;
     int   nBaseShift;
     int   nPwr10;
} NUMPARSE;
```

The *cDig* element is set to the size of the *rgbDi*g array, and *dwInFlags* is set to parsing options. All other elements may be uninitialized and are set by the function, except on error, as described in the following paragraphs. The *cDig* element is also modified by the function to reflect the actual number of digits written to the *rgbDig* array.

The *cchUsed* element of the NUMPARSE sturcture is filled in with the number of characters (from the beginning of the string) that were successfully parsed. This allows the caller to determine if the entire string was part of the number (as required by functions such as **VarI2FromStr**), or where to continue parsing the string.

The *nBaseShift* element gives the number of bits per digit (3 or 4 for octal and hexadecimal numbers, and zero for decimal).

The following apply only to decimal numbers:

- *nPwr10* sets the decimal point position by giving the power of 10 of the least significant digit.
- If the number is negative, NUMPRS_NEG will be set in *dwOutFlags*.
- If there are more non-zero decimal digits than will fit into the digit array, the NUMPRS_INEXACT flag will be set.

VarParseNumFromNum

HRESULT VarNumFromParseNum(

[in] NUMPARSE ***pnumprs**,,
[in] unsigned char **rgbDig*,
[in] unsigned long *dwVtBits*,
[out] VARIANT **pvar*
);

Once the number is parsed, the caller can call **VarNumFromParseNum()** to convert the parse results to a number. The NUMPARSE structure and digit array must be passed in unchanged from the **VarParseNumFromStr()** call. This function will choose the smallest type allowed that can hold the result value with as little precision loss as possible. The result variant is an [out] parameter, so its contents are not freed before storing the result.

Parameters

pnumprs

Parsed results.

rgbDig

Array.

dwVtBits

Contains one bit set for each type that is acceptable as a return value (in many cases, just one bit).

Pvar

Result variant.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
DISP_E_OVERFLOW	The number is too large to be represented in an allowed type. There is no error if precision is lost in the conversion.

The *rgbDig* array is filled in with the values for the digits in the range 0-7, 0-9, or 0-15, depending on whether the number is octal, decimal, or hexadecimal. All leading zeros have been stripped off. For decimal numbers, trailing zeros are also stripped off, unless the number is zero, in which case a single zero digit will be present.

For rounding decimal numbers, the digit array must be at least one digit longer than the maximum required for data types. The maximum number of digits required for the DECIMAL data type is 29, so the digit array must have room for 30 digits. There must also be enough digits to accept the number in octal, if that parsing options is selected. (Hexadecimal and octal numbers are limited by **VarNumFromParseNum()** to the magnitude of an **unsigned long** [32 bits], so they need 11 octal digits.)

Date and Time Conversion Functions

The following functions are provided by Oleauto32.dll (for 32-bit systems) and Ole2disp.dll (for 16-bit systems) to convert between dates and times stored in MS-DOS format and the variant representation.

DosDateTimeToVariantTime Quick Info

```
int DosDateTimeToVariantTime(
  unsigned short   wDOSDate,
  unsigned short   wDOSTime,
  double FAR*   pvtime
);
```

Converts the MS-DOS representation of time to the date and time representation stored in a variant.

Parameters

wDOSDate

The MS-DOS date to convert.

wDOSTime

The MS-DOS time to convert.

pvtime

Pointer to the location to store the converted time.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Result	Meaning
True	Success.
False	Failure.

Comments

MS-DOS records file dates and times as packed 16-bit values. An MS-DOS date has the following format.

Bits	Contents
0-4	Day of the month (1-31).
5-8	Month (1 = January, 2 = February, and so on).
9-15	Year offset from 1980 (add 1980 to get the actual year).

An MS-DOS time has the following format.

Bits	Contents
0-4	Second divided by 2.
5-10	Minute (0-59).
11-15	Hour (0- 23 on a 24-hour clock).

VariantTimeToDosDateTime Quick Info

int VariantTimeToDosDateTime(

double *vtime*,
unsigned short FAR* *pwDOSDate*,
unsigned short FAR* *pwDOSTime*
);

Converts the variant representation of a date and time to MS-DOS date and time values.

Parameters

vtime

The variant time to convert.

pwDOSDate

Pointer to the location to store the converted MS-DOS date.

pwDOSTime

Pointer to the location to store the converted MS-DOS time.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Result	Meaning
True	Success.
False	Failure.

Comments

A variant time is stored as an 8-byte real value (**double**), representing a date between January 1, 1753 and December 31, 2078, inclusive. The value 2.0 represents January 1, 1900; 3.0 represents January 2, 1900, and so on. Adding 1 to the value increments the date by a day. The fractional part of the value represents the time of day. Therefore, 2.5 represents noon on January 1, 1900; 3.25 represents 6:00 A.M. on January 2, 1900, and so on. Negative numbers represent the dates prior to December 30, 1899.

For a description of the MS-DOS date and time formats, see **DosDateTimeToVariantTime**.

VariantTimeToSystemTime

int VariantTimeToSystemTime(

double *vtime,*
SYSTEMTIME **psystime*
);

Converts the variant representation of time-to-system time values.

Parameters

vtime

The variant time that will be converted.

psystime

Pointer to the location where the converted time will be stored.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Result	Meaning
True	Success.
False	Failure.

Comments

A variant time is stored as an 8-byte real value (**double**), representing a date between January 1, 1753 and December 31, 2078, inclusive. The value 2.0 represents January 1, 1900; 3.0 represents January 2, 1900, and so on. Adding 1 to the value increments the date by a day. The fractional part of the value represents the time of day. Therefore, 2.5 represents noon on January 1, 1900; 3.25 represents 6:00 A.M. on January 2, 1900, and so on. Negative numbers represent the dates prior to December 30, 1899.

Using the SYSTEMTIME structure is useful because:

- It spans all time/date periods. MS-DOS date/time is limited to representing only those dates between 1/1/1980 and 12/31/2107.
- The date/time elements are all easily accessible without needing to do any bit decoding.
- The National Language Support data and time formating functions **GetDateFormat()** and **GetTimeFormat()** take a SYSTEMTIME value as input.
- It is the default Win32 time and date data format supported by Windows NT and Windows 95.

SystemTimeToVariantTime

int SystemTimeToVariantTime(*psystime*, *pvtime***)**
SYSTEMTIME ***psystime***
double **vtime*

Converts the variant representation of time-to-system time values.

Parameters

psystime

The system time.

vtime

Returned variant time.

Return Value

The return value obtained from the returned HRESULT is one of the following.

Result	Meaning
True	Success.
False	Failure.

Comments

A variant time is stored as an 8-byte real value (**double**), representing a date between January 1, 1753 and December 31, 2078, inclusive. The value 2.0 represents January 1, 1900; 3.0 represents January 2, 1900, and so on. Adding 1 to the value increments the date by a day. The fractional part of the value represents the time of day. Therfore, 2.5 represents noon on January 1, 1900; 3.25 represents 6:00 A.M. on January 2, 1900, and so on. Negative numbers represent the dates prior to December 30, 1899.

The SYSTEMTIME structure is useful for the following reasons:

- It spans all time/date periods. MS-DOS date/time is limited to representing only those dates between 1/1/1980 and 12/31/2107.
- The date/time elements are all easily accessible without needing to do any bit decoding.
- The National Data Support data and time formating functions **GetDateFormat()** and **GetTimeFormat()** take a SYSTEMTIME value as input.
- It is the default Win32 time/date data format supported by Windows NT and Windows 95.

Type Libraries and the Object Description Language

When you expose ActiveX objects, it allows interoperability with the programs of other vendors. For vendors to use these objects, they must have access to the characteristics of the objects (properties and methods). To make this information available developers must:

- Publish object and type definitions (for example, as printed documentation).
- Code objects into a compiled .c or .cpp file so they can be accessed using **IDispatch::GetTypeInfo** or implementations of the **ITypeInfo** and **ITypeLib** interfaces.
- Use the MIDL compiler or the MkTypLib utility to create a type library that contains the objects, and then make the type library available.

The Microsoft Interface Definition Language (MIDL) compiler and the MkTypLib utility both compile scripts that are written in the Object Description Language (ODL). Microsoft has expanded the Interface Definition Language (IDL) to contain the complete ODL syntax. You should use the MIDL compiler in preference to MkTypLib, since MkTypLib is being phased out and will no longer be supported.

For more information about the MIDL compiler, refer to the *MIDL Programmer's Guide and Reference* in the Win32 Software Development Kit (SDK).

The following descriptions and references are contained in this chapter:

- Contents of a type library
- Using MIDL and MkTypeLib
- MkTypLib type library creation
- ODL file syntax
- ODL reference

Contents of a Type Library

Type libraries are compound document files (.tlb files) that include information about types and objects exposed by an ActiveX application. A type library can contain any of the following:

- Information about data types, such as aliases, enumerations, structures, or unions.
- Descriptions of one or more objects, such as a module, interface, **IDispatch** interface (dispinterface), or component object class (coclass). Each of these descriptions is commonly referred to as a *typeinfo.*
- References to type descriptions from other type libraries.

By including the type library with a product, the information about the objects in the library can be made available to the users of the applications and programming tools. Type libraries can be shipped in any of the following forms:

- A resource in a dynamic link library (DLL). This resource should have the type TypeLib and an integer ID. It must be declared in the resource (.rc) file as follows:

  ```
  1 typelib mylib1.tlb
  2 typelib mylib2.tlb
  ```

  There can be multiple type library resources in a DLL. Application developers should use the resource compiler to add the .tlb file to their own DLL. A DLL with one or more type library resources typically has the file extension .olb (object library).
- A resource in an .exe file. The file can contain multiple type libraries.
- A stand-alone binary file. The .tlb (type library) file output by the MkTypLib utility is a binary file.

Object browsers, compilers, and similar tools access type libraries through the interfaces **ITypeLib**, **ITypeInfo**, and **ITypeComp**. Type library tools (such as MkTypLib) can be created using the interfaces **ICreateTypeLib** and **ICreateTypeInfo**.

Using MIDL and MkTypeLib

Files parsed by MkTypeLib are .odl files. Files parsed by MIDL are referred to as .idl files, although they can contain the same syntax elements as .odl files. The MIDL compiler and the MkTypeLib utility both compile scripts wriitten in the Object Description Language (ODL). However, MkTypLib is obsolete and you should use the MIDL compiler instead.

Adding ODL to an IDL Definition

The .ODL files provide object definitions that are added to the type descriptions in a type library. The MkTypeLib .EXE parses files written in the ODL syntax, generates the type libraries, and optionally creates C++ header files that contain the same definitions.

The top-level element of the ODL syntax is the **library** statement (or library block). Every other ODL statement (with the exception of the attributes that can be applied to the **library** statement) must be defined in the library block.

The MIDL function generates a type library when it sees a **library** statement in the same way that MkTypeLib does. The statements found in the library block follow essentially the same syntax as earlier versions of the ODL language.

ODL attributes can be applied to an element both inside and outside of the library block. Outside the block, they typically do nothing, unless the element is referenced from within the block by using it as a base type, inheriting from it, or referencing it on a line such as this:

```
library a
{
      interface [xyz]];
      struct bar;
      ...
};
```

If an element defined outside of the block is referenced in the block, its definition is put into the generated type library.

Anything outside of the library block is an .idl file, and the MIDL compiler processes it as usual. Typically, this means generating remote stubs for it.

Support for ODL Base Types

There are a number of base types supported by MkTypLib that are not directly supported by MIDL. The MIDL function gets its definitions for these base types by automatically importing Oleauto.idl, and Oleidl.idl whenever it encounters a library statement. This means that Oleauto.idl and Oleidl.idl (along with the imported Unknwn.idl and Wtypes.idl files) must be somewhere in the user's INCLUDE path. The OLE and Automation DLLs must also be in the system if the user compiles an .idl file that contains a library statement.

The following table contains the command line options for the MIDL compiler:

Option	Description
/**tlb** <*filename*>	Name of the type library file name for the output .tlb file. If not specified, this is the same as the name of the .odl file with the extension of .tlb.
/**h** <*filename*>	Similar to the **/header** in MIDL, but also containing definitions related to the type library. For information on the **/header** switch see Chapter …
/ <*system*>	<*system*> is win16, win32, mac, mips, alpha, ppc, or ppc32.
/**align** <#>	Same as the **/Zp** switch in MIDL. It sets the alignment for types in the library. For more information on the **/Zp** switch see…
/**o** <*output file*>	Redirects output.
/**nologo**	Disables the display of the copyright logo.
/**nocpp**	Same as the **/no_cpp** switch in MIDL. For more information on the **/no_cpp** switch see…
/**mktyplib203**	Puts MIDL into MkTypLib-compatibility mode.

/mktyplib203 Option

The MIDL compiler behaves differently from the MkTypLib utility. The /**mktyplib203** option removes most of these differences and makes MIDL act like MkTypLib version 2.03.

For example, BOOL (a MkTypLib base type) is defined differently in MIDL than it is in MkTypLib. MkTypLib treats BOOL as a VARIANT_BOOL. However, BOOL is defined in the file Wtypes.idl as a long data type. If a VARIANT_BOOL is to be placed in the type library, it has to explicitly use VARIANT_BOOL in the IDL/ODL file. If BOOL is used when VARIANT_BOOL is meant to be used, then the /**mktyplib203** option should also be used.

MIDL normally puts globally unique identifier (GUID) predefinitions in its generated header files, and only puts GUID instantiations in the file generated by the **/iid** option. With the /**mktyplib203** option, MIDL defines GUIDs in the header files in the way that MkTypLib does. They are defined with a macro that can be compiled conditionally to generate either a predefined or an instantiated GUID.

With the /**mktyplib203** option enabled, it is invalid to put any statements outside of the library block. A pure ODL syntax must be used; it cannot be mixed and matched in this mode.

MkTypLib is used to require **structure, union,** and **enum t**o be defined as part of type definitions. For example:

```
typedef struct foo { int i; } bar;
```

In this statement, MkTypLib generates a TKIND_RECORD named "bar." Because the "foo" was not recorded anywhere in the type library, it can be omitted.

MIDL allows normal C definitions of **structure, union,** and **enum**:

```
struct foo {int i;};
     typedef struct foo bar;
```

- Or -

```
     typedef struct foo {int i;} bar;
```

This statement generates a TKIND_RECORD named "foo" and (if the type definition is public) a TKIND_ALIAS named "bar." The "foo" can still be omitted, in which case MIDL generates a name for it.

When the /**mktyplib203** option is enabled, the original MkTypLib type definition syntax is required for structures, unions, and enumerators. The behavior is the same as under MkTypLib (that is, "foo" is not included in the type library).

Note MkTypLib permits some scoping errors, such as giving enumerators their own scope. These errors are fixed by MIDL, and cannot be reintroduced, even with the **/mktyplib203** switch. Even though the /**mktyplib203** switch enables MIDL to compile most earlier .odl files, there can be a few exceptions. These are cases where the .odl files were already broken, and MkTypLib did not catch the errors.

MkTypLib: Type Library Creation Tool

MkTypLib processes scripts written in the Object Description Language (ODL), producing a type library and an optional C or C++ header file.

MkTypLib uses the **ICreateTypeLib** and **ICreateTypeInfo** interfaces to create type libraries. Type libraries can then be accessed by tools, such as type browsers and compilers that use the **ITypeLib** and **ITypeInfo** interfaces, as shown in the following figure.

{ewc msdncd, EWGraphic, bsd23528 0 /a "SDK_02.WMF"}

Invoking MkTypLib

To invoke MkTypLib, click the Windows® 95 **Start** button, and then click **Run**. Enter the following command line in the **Open** box:

MkTypLib [*options*] *ODLfile*

MkTypLib creates a type library (.tlb) file based on the object description script in the file specified by *ODLfile*. It can optionally produce a header (.h) file, which is a stripped version of the input file. This file is included in C or C++ programs that want to access the types defined in the input file. In the header file, MkTypLib inserts DEFINE_GUID macros for each element defined in the type library (such as interface**,** dispinterface**,** and so on).

There can be a series of options, each prefixed with a hyphen (-) or a slash (/), as follows:

Option	Description
/? or **/help**	Displays command line Help. In this case, *ODLfile* does not need to be specified.
/align:*alignment*	Sets the default alignment for types in the library. An *alignment* value of 1 indicates natural alignment; *n* indicates alignment on byte *n*.
/cpp_cmd *cpppath*	Specifies *cpppath* as the command to run the C preprocessor. By default, MkTypLib invokes CL.
/cpp_opt "*options*"	Specifies options for the C preprocessor. The default is **/C /E /D__MkTypLib__**.
/D *define*[=*value*]	Defines the name *define* for the C preprocessor. The *value* is its optional value. No space is allowed between the equal sign (=) and the value.
/h *filename*	Specifies *filename* as the name for a stripped version of the input file. This file can be used as a C or C++ header file.
/I *includedir*	Specifies *includedir* as the directory where include files are located for the C preprocessor.
/nocpp	Suppresses invocation of the C preprocessor.
/nologo	Disables the display of the copyright banner.
/o *outputfile*	Redirects output (for example, error messages) to the specified *outputfile.*
/tlb *filename*	Specifies *filename* as the name of the output .tlb file. If not specified, it will be the same name as the *ODLfile*, with the extension .tlb.
/win16 /win32 /mac /mips /alpha /ppc /ppc32	Specifies the output type library to be produced. The default is the current operating system.
/w0	Disables warnings.

Although MkTypLib offers minimal error reporting, error messages include accurate line number and column number information that can be used with text editors to locate the source of errors.

MkTypLib spawns the C preprocessor. The symbol **__MKTYPLIB__** is predefined for the preprocessor.

ODL File Syntax

The general syntax for an .odl file is as follows:

[*attributes***] library** *libname* **{***definitions***};**

The *attributes* associate characteristics with the library, such as its Help file and universally unique identifier (UUID). Attributes must be enclosed in square brackets.

The *definitions* consist of the descriptions of the imported libraries, data types, modules, interfaces, dispinterfaces, and coclasses that are part of the type library. Braces ({}) must surround the definitions.

The following table summarizes the elements that can appear in *definitions*. Each element is described in more detail later in this chapter, in the section "ODL Reference."

Purpose	Library element	Description
Allows references to other type libraries.	**importlib** (*lib1*)	Specifies an external type library that contains definitions that are referenced in this type library.
Declares data types used by the objects in this type library.	**typedef [***attributes***]** *aliasname*	An alias declared using C syntax. Must have at least one *attribute* to be included in the type library.
	typedef [*attributes*] **enum**	An enumeration declared using the C keywords **typedef** and **enum**.
	typedef [*attributes*] **struct**	A structure declared using the C keywords **typedef** and **struct**.
	typedef [*attributes*] **union**	A union declared using the C keywords **typedef** and **union**.
Describes functions that enable querying the DLL.	[*attributes*] **module**	Constants and general data functions whose actions are not restricted to any specified class of objects.
Describes interfaces.	[*attributes*] dispinterface	An interface describing the methods and properties for an object that must be accessed through **IDispatch::Invoke**.
	[*attributes*] interface	An interface describing the methods and properties for an object that can be accessed either through **IDispatch::Invoke** or through VTBL entries.
Describes OLE classes.	[*attributes*] **coclass**	Specifies a top-level object with all of its interfaces and dispinterfaces.

In the library description, modules, interfaces, dispinterfaces, and coclasses follow the same general

syntax:

[*attributes***]** *elementname typename* **{**
memberdescriptions
};

The *attributes* set characteristics for the element. The *elementname* is a keyword that indicates the kind of item (module, interface, dispinterface, or coclass), and the *typename* defines the name of the item. The *memberdescriptions* define the members (constants, functions, properties, and methods) of each element.

Aliases, enumerations, unions, and structures have the following syntax:

typedef [*typeattributes***]** *typekind typename* **{**
memberdescriptions
};

For these types, the attributes follow the **typedef** keyword, and the *typekind* indicates the data type (**enum**, **union**, or **struct**). For details, see "Attribute Descriptions" later in this chapter.

Note The square brackets ([ ])and braces ({ }) in these descriptions are part of the syntax, and are not descriptive symbols. The semicolon after the closing brace (**}**) that terminates the library definition (and all other type definitions) is optional.

ODL File Example

The following example shows the .odl file for the Lines sample file, extracted from Lines.odl:

```
[
     uuid(3C591B20-1F13-101B-B826-00DD01103DE1),           // LIBID_Lines.
     helpstring("Lines 1.0 Type Library"),
     lcid(0x09),
     version(1.0)
]
library Lines
{
     importlib("stdole.tlb");
     #define DISPID_NEWENUM -4

     [
           uuid(3C591B25-1F13-101B-B826-00DD01103DE1),  // IID_Ipoint.
           helpstring("Point object."),
           oleautomation,
           dual
     ]
     interface IPoint : IDispatch
     {
           [propget, helpstring("Returns and sets x coordinate.")]
           HRESULT x([out, retval] int* retval);
           [propput, helpstring("Returns and sets x coordinate.")]
           HRESULT x([in] int Value);

           [propget, helpstring("Returns and sets y coordinate.")]
           HRESULT y([out, retval] int* retval);
           [propput, helpstring("Returns and sets y coordinate.")]
           HRESULT y([in] int Value);
     }

// Additional interfaces omitted for brevity.

     [
           uuid(3C591B27-1F13-101B-B826-00DD01103DE1),           //
IID_Ipoints.
           helpstring("Points collection."),
           oleautomation,
           dual
     ]
     interface IPoints : IDispatch
     {
           [propget, helpstring("Returns number of points in collection.")]
           HRESULT Count([out, retval] long* retval);

           [propget, id(0),
           helpstring("Given an index, returns a point in the collection.")]
           HRESULT Item([in] long Index, [out, retval] IPoint** retval);

           [propget, restricted, id(DISPID_NEWENUM)] // Must be propget.
           HRESULT _NewEnum([out, retval] IUnknown** retval);
```

```
    }

// Additional interface omitted for brevity.

    [
        uuid(3C591B22-1F13-101B-B826-00DD01103DE1),     //
IID_Iapplication
        helpstring("Application object."),
        oleautomation,
        dual
    ]
    interface IApplication : IDispatch
    {
        [propget, helpstring("Returns the application of the object.")]
        HRESULT Application(    [out, retval] IApplication** retval);

        [propget,
        helpstring("Returns the full name of the application.")]
        HRESULT FullName([out, retval] BSTR* retval);
        [propget, id(0),
        helpstring("Returns the name of the application.")]
        HRESULT Name([out, retval] BSTR* retval);

        [propget, helpstring("Returns the parent of the object.")]
        HRESULT Parent([out, retval] IApplication** retval);

        [propput]
        HRESULT Visible([in] boolean VisibleFlag);
        [propget, helpstring
        ("Sets or returns whether the main window is visible.")]
        HRESULT Visible(  [out, retval] boolean* retval);

        [helpstring("Exits the application.")]
        HRESULT Quit();

// Additional methods omitted for brevity.

        [helpstring("Creates new Point object initialized to (0,0).")]
        HRESULT CreatePoint(    [out, retval] IPoint** retval);
    }

    [
        uuid(3C591B21-1F13-101B-B826-00DD01103DE1),     // CLSID_Lines.
        helpstring("Lines Class"),
        appobject
    ]
    coclass Lines
    {
        [default] interface IApplication;
            interface IDispatch;
    }
}
```

The example describes a library named Lines that imports the standard OLE library Stdole.tlb. The **#define** directive defines the constant DISPID_NEWENUM, which is needed for the **_NewEnum** property

of the **IPoints** collection.

The example shows declarations for three interfaces in the library: **IPoint**, **IPoints**, and **IApplication**. Because all three are dual interfaces, their members can be invoked through **IDispatch** or directly through VTBLs. In addition, all of their members return HRESULT values and pass their return values as **retval** parameters. Therefore, they can support the **IErrorInfo** interface, through which they can return detailed error information in whatever way they are invoked.

The **IPoint** interface has two properties, **X** and **Y**, and two pairs of accessor functions to get and set the properties.

The **IPoint** interface is a collection of points. It supports three read-only properties, each of which has a single accessor function. The **Count** and **Item** properties return the number of points and the value of a single point, respectively. The **_NewEnum** property, required for collection objects, returns an enumerator object for the collection. This property has the **restricted** attribute, indicating that it should not be invoked from a macro language.

The **IApplication** interface describes the application object. It supports the properties **Application, FullName, Name, Parent, Visible,** and **Pane**. It supports the methods **Quit, CreateLine,** and **CreatePoint**.

Finally, the script defines a coclass named Lines. The **appobject** attribute makes the members of the coclass (**IApplication** and **IDispatch**) globally accessible in the type library. **IApplication** is defined as the **default** member, indicating that it is the programmability interface intended for use by macro languages.

Source File Contents

The following sections describe the proper format for comments, constants, identifiers, and other syntactic items in an .odl file.

Array Definitions

MkTypLib accepts both fixed-size arrays and arrays declared as SAFEARRAY.

Use a C-style syntax for a fixed size array:

*type arrname***[***size***];**

To describe a SAFEARRAY, use the following syntax:

SAFEARRAY (*elementtype***) ****arrayname***

A function returning a SAFEARRAY has the following syntax:

SAFEARRAY (*elementtype*) *myfunction***(***parameterlist***);**

Comments

To include comments in an .odl file, use a C-style syntax in either block form (/*...*/) or single-line form (//). MkTypLib ignores the comments, and does not preserve them in the header (.h) file.

Constants

A constant can be either numeric or a string, depending on the attribute.

Numeric

Numeric input is usually an integer (in either decimal or in hexadecimal, using the standard 0x format), but can also be a single character constant (for example, \0).

String

A string is delimited by double quotation marks (") and cannot span multiple lines. The backslash character (\) acts as an escape character. The backslash character followed by any character (even another backslash) prevents the second character from being interpreted with any special meaning. The backslash is not included in the text.

For example, to include a double quotation mark (") in the text without causing it to be interpreted as the closing delimiter, it should be preceded with a backslash (\"). Similarly, a double backslash (\\) should be used to put a backslash into the text. Some examples of valid strings are:

```
"commandName"
"This string contains a \"quote\"."
"Here's a pathname: c:\\bin\\binp"
```

A string can be up to 255 characters long.

File Names

A file name is a string that represents either a full or partial path. Automation expects to find files in directories that are referenced by the type library registration entries, so partial path names are typically used. For more information about registration, refer to Chapter 2, "Exposing ActiveX objects."

Forward Declarations

Forward declarations permit forward references to types. Forward references have the following form:

```
typedef struct mydata;
interface aninterface;
dispinterface fordispatch;
coclass pococlass;
```

Globally Unique ID (GUID)

A universally unique ID (UUID) is a globally unique ID (GUID). This number is created by running the Guidgen.exe command line program. Guidgen.exe never produces the same number twice, no matter how many times it is run or how many different machines it runs on. Every entity that needs to be uniquely identified (such as an interface) has a globally unique ID.

Identifiers

Identifiers can be up to 255 characters long, and must conform to C-style syntax. MkTypLib is case sensitive, but it generates type libraries that are case insensitive. It is therefore possible to define a user-defined type whose name differs from that of a built-in type only by case. User-defined type names (and member names) that differ only in case refer to the same type or member. Except for property accessor functions, it is invalid for two members of a type to have the same name, regardless of case.

Intrinsic Data Types

The following data types are recognized by MkTypLib:

Type	Description
boolean	Data item that can have the value True or False. The size maps to VARIANT_BOOL.
char	8-bit signed data item.
double	64-bit IEEE floating-point number.
int	Signed integer, whose size is system dependent.
float	32-bit IEEE floating point number.
long	32-bit signed integer.
short	16-bit signed integer.
wchar_t	Unicode character accepted only for 32-bit type libraries.
BSTR	Length-prefixed string, as described in Chapter 5, "Dispatch Interface and API Functions."
CURRENCY	8-byte, fixed-point number.
DATE	64-bit floating-point fractional number of days since December 30, 1899.
SCODE	Built-in error type that corresponds to VT_ERROR. An SCODE (used in 16-bit systems only) does not contain the additional error information provided by HRESULT.
VARIANT	One of the variant data types as described in Chapter 5, "Dispatch Interface and API Functions."
IDispatch *	Pointer to the **IDispatch** interface.
IUnknown *	Pointer to the **IUnknown** interface. (Any OLE interface can be represented by its **IUnknown** interface.)
SAFEARRAY(TypeName)	**TypeName** is any of the above types. Array of these types.
TypeName*	**TypeName** is any of the above types. Pointer to a type.
void	Allowed only as return type for a function, or in a function parameter list to indicate no arguments.
HRESULT	Return type used for reporting error information in interfaces, as described in *Microsoft OLE Programmer's Guide and Reference*.
LPWSTR	Unicode string accepted only for 32-bit type libraries.
LPSTR	Zero-terminated string.

Not all of the above types can be marshaled by Automation to another process or thread. The list of types that can be marshaled are called Automation compatible types, and are listed under the **oleautomation** attribute description.

The keyword **unsigned** can be specified before **int, char, short,** and **long**.

String Definitions

Strings can be declared using the LPSTR data type, which indicates a zero-terminated string, and with the BSTR data type, which indicates a length-prefixed string (as defined in Chapter 5, "Dispatch Interface and API Functions"). In 32-bit type libraries, Unicode strings can be defined with the LPWSTR data type.

ODL Reference

This section provides reference material on the attributes, statements, and directives that are part of the Object Description Language (ODL).

Attribute Descriptions

The following sections describe the ODL attributes and the types of objects that they apply to, along with the equivalent flags set in the object's type information.

appobject

Description

Identifies the Application object.

Allowed on

Coclass.

Comments

Indicates that the members of the class can be accessed without qualification when accessing this type library.

Flags

TYPEFLAG_FAPPOBJECT

aggregatable

Description

Indicates that the class supports aggregation.

Allowed on

Coclass.

Comments

Indicates that the members of the class can be aggregated.

Flags

TYPEFLAG_FAGGREGATABLE

Example

```
[    uuid(1e196b20-1f3c-1069-996b-00dd010fe676),
     aggregatable
]
coclass Form
{
     [default] interface IForm;
     [default, source] interface IFormEvents;
}
```

bindable

Description

Indicates that the property supports data binding.

Allowed on

Property.

Comments

Refers to the property as a whole, so it must be specified wherever the property is defined. The attribute should be specified on both the property get description and the property set description.

Flags

FUNCFLAG_FBINDABLE, VARFLAG_FBINDABLE

control

Description

Indicates that the item represents a control from which a container site will derive additional type libraries or coclasses.

Allowed on

Type libraries, coclasses.

Comments

This attribute allows type libraries that describe controls to be marked so that they are not be displayed in type browsers intended for nonvisual objects.

Flags

TYPEFLAG_FCONTROL, LIBFLAG_FCONTROL

custom(<*guid*>, <*value*>)

Description

Indicates a custom attribute (one not defined by Automation). This feature enables the independent definition and use of attributes.

<*guid*> the standard GUID form

<*value*>a value that can be put into a variant. (See also the **Const** directive.)

Allowed on

Library, typeinfo, typlib, variable, function, parameter.

Not allowed on

A member of a coclass (IMPLTYPE).

Representation

Can be retrieved using:

ITypeLib2::GetCustData
ITypeInfo2::GetCustData
ITypeLib2::GetFuncCustData
ITypeLib2::GetVarCustData
ITypeLib2::GetParamCustData

Example

The following example shows how to add a string-valued attribute that gives the programmatic ID (ProgID) for a class:

```
[
     custom(GUID_PROGID, "DAO.Dynaset")
]
coclass Dynaset
{
     [default] interface Dynaset;
     [default, source] interface IDynasetEvents;
}
```

default

Description

Indicates that the interface or dispinterface represents the default programmability interface. Intended for use by macro languages.

Allowed on

Coclass member.

Comments

A coclass can have two **default** members at most. One represents the source interface or dispinterface, and the other represents the sink interface or dispinterface. If the **default** attribute is not specified for any member of the coclass or cotype, the first source and sink members that do not have the **restricted** attribute will be treated as the defaults.

Flags

IMPLTYPEFLAG_FDEFAULT

defaultbind

Description

Indicates the single, bindable property that best represents the object.

Allowed on

Property.

Comments

Properties that have the **defaultbind** attribute must also have the **bindable** attribute. The **defaultbind** attribute cannot be specified on more than one property in a dispinterface.

This attribute is used by containers that have a user model that involves binding to an object rather than binding to a property of an object. An object can support data binding and not have this attribute.

Flags

FUNCFLAG_FDEFAULTBIND, VARFLAG_FDEFAULTBIND

defaultcollelem

Description

Allows for optimization of code.

Allowed on

Type information, property.

Comments

In Visual Basic for Applications (VBA), "foo!bar" is normally syntactic shorthand for foo.defaultprop("bar"). Because such a call is significantly slower than accessing a data member of foo directly, an optimization has been added in which the compiler looks for a member named "bar" on the type of foo. If such a member is found and flagged as an accessor function for an element of the default collection, a call is generated to that member function. To allow vendors to produce object servers that will be optimized in this way, the member flag should be documented.

Because this optimization searches the type of item that precedes the '!', it will optimize calls of the form MyForm!bar only if MyForm has a member named "bar," and it will optimize MyForm.Controls!bar only if the return type of Controls has a member named **bar**. Even though MyForm!bar and MyForm.Controls!bar both would normally generate the same calls to the object server, optimizing these two forms requires that the object server add the **bar** method in both places.

Use of [defaultcollitem] must be consistent for a property. For example, if it is present on a **Get**, it must also be present on a **Put**.

Flags

FUNCFLAG_FDEFAULTCOLLELEM, VARFLAG_FDEFAULTCOLLELEM

Example

A form has a button on it named **Button1**. User code can access the button using property syntax or ! syntax, as shown below.

```
Sub Test()
     Dim f As Form1
     Dim b1 As Button
     Dim b2 As Button

     Set f = Form1

     Set b1 = f.Button1            ' Property syntax
     Set b = f!Button1       ' ! syntax
End Sub
```

To use the property syntax and the ! syntax properly, see the form in the type information below.

```
     [     odl,
           dual,
           uuid(1e196b20-1f3c-1096-996b-00dd010ef676),
           helpstring("This is IForm"),
           restricted
     ]
     interface IForm1: IForm
     {
```

```
        [propget, defaultcollelem]
        HRESULT Button1([out, retval] Button *Value);
}
```

defaultvalue(*value*)

Description

Enables specification of a default value for a typed optional parameter.

Allowed on

Parameter.

Comments

The expression *value* resolves to a constant that can be described in a variant. The ODL already allows some expression forms, as when a constant is declared in a module. The same expressions are supported without modification.

The following example shows some legal parameter descriptions:

```
interface IFoo
{
     void Ex1([defaultvalue(44)] LONG    i);
     void Ex2([defaultvalue(44)] SHORT   i);
     void Ex3([defaultvalue("Hello")] BSTR     i);
}
```

The following rules apply:

1. It is invalid to specify a default value for a parameter whose type is a safe array. It is invalid to specify a default value for any type that cannot go in a variant, including structures and arrays.
2. Optional parameters must follow default value parameters. Optional parameters and default value parameters must follow mandatory parameters.
3. The default value can be any constant that is represented by a VARIANT data type.

Flags

none

Example

```
interface QueryDef
{

// Type is now known to be a LONG type (good for browser in VBA and
// for a C/C++ programmer) and also has a default value of
// dbOpenTable (constant).

HRESULT    OpenRecordset(    [in, defaultvalue(dbOpenTable)]
     LONG Type,

     [out,retval]
Recordset **pprst);
}
```

defaultvtbl

Description

Enables an object to have two different source interfaces.

The **default** interface is an interface or dispinterface that is the default source interface. If the interface is a:

- **dual** interface, sinks receive events through **IDispatch**.
- **VTBL** interface, event sinks receive events through VTBL.
- **dispinterface,** sinks receive events through **IDispatch**.
- **defaultvtable,** a default VTBL interface, which cannot be a dispinterface – it must be a dual, VTBL, or interface. If the interface is a dual interface, then sinks receive events through the VTBL.

An object can have both a default source and a default VTBL source interface with the same interface ID (IID or GUID). In this case, a sink should advise using IID_IDISPATCH to receive dispatch events, and use the specific interface ID to receive VTBL events.

Allowed on

A member of a coclass.

Comments

For normal (non-source) interfaces, an object can support a single interface that satisfies consumers who want to use **IDispatch** access as well as VTBL access (a dual interface). Because of the way source interfaces work, it is not possible dual interface for source interfaces. The object with the source interface is in control of whether calls are made through **IDispatch** or through the VTBL. The sink does not provide any information about how it wants to receive the events. The only action that object-sourcing events can take would be to use the least common denominator, the **IDispatch** interface. This effectively reduces a dual interface to a dispatch interface with regard to sourcing events.

Interface	Flag it translates into
default	IMPLTYPEFLAG_FDEFAULT
default, **source**	IMPLTYPEFLAG_FDEFAULT
	IMPLTYPEFLAG_FSOURCE
defaultvtable, **source**	IMPLTYPEFLAG_FDEFAULT
	IMPLTYPEFLAG_FDEFAULTVTABLE
	IMPLTYPEFLAG_FSOURCE

Flags

IMPLTYPEFLAG_FDEFAULTVTABLE. If this flag is set, then IMPLTYPEFLAG_FDEFAULT is also set.

Example

```
[    odl,
     dual,
     uuid(1e196b20-1f3c-1069-996b-00dd010ef676),
     restricted
]
```

```
interface IForm: IDispatch
{
     [propget]
     HRESULT Backcolor([out, retval] long *Value);

     [propput]
     HRESULT Backcolor([in] long Value);


     [propget]
     HRESULT Name([out, retval] BSTR *Value);

     [propput]
     HRESULT Name([in] BSTR Value);
}

[    odl,
     dual,
     uuid(1e196b20-1f3c-1069-996b-00dd010ef767),
     restricted
]
interface IFormEvents: IDispatch
{
     HRESULT Click();
     HRESULT Resize();
}

[uuid(1e196b20-1f3c-1069-996b-00dd010fe676)]
coclass Form
{
     [default] interface IForm;
     [default, source] interface IFormEvents;
     [defaultvtable, source] interface IFormEvents;
}
```

displaybind

Description

Indicates that a property should be displayed as bindable to the user.

Allowed on

Property.

Comments

Properties that have the **displaybind** attribute must also have the **bindable** attribute. An object can support data binding and not have this attribute.

Flags

FUNCFLAG_FDISPLAYBIND, VARFLAG_FDISPLAYBIND

dllname(*str*)

Description

Defines the name of the DLL that contains the entry points for a module.

Allowed on

Module (required).

Comments

The *str* argument gives the file name of the DLL.

dual

Description

Identifies an interface that exposes properties and methods through **IDispatch** and directly through the VTBL.

Allowed on

Interface.

Comments

The interface must be compatible with Automation and derive from **IDispatch**. Not allowed on dispinterfaces.

The **dual** attribute creates an interface that is both a **Dispatch** interface and a Component Object Model (COM) interface. The first seven entries of the VTBL for a dual interface are the seven members of **IDispatch**, and the remaining entries are COM entries for direct access to members of the dual interface. All of the parameters and return types specified for members of a dual interface must be compatible with Automation types.

All members of a dual interface must pass an HRESULT as the function's return value. Members that need to return other values should specify the last parameter as [**retval**, **out**] indicating an output parameter that returns the value of the function. In addition, members that need to support multiple locales should pass an *lcid* parameter.

A dual interface provides for both the speed of direct VTBL binding and the flexibility of **IDispatch** binding. For this reason, dual interfaces are recommended whenever possible.

> **Note** If an application accesses object data by casting the THIS pointer in the interface call, the VTBL pointers in the object should be checked against the VTBL pointers to ensure that they are connected to the appropriate proxy.

Specifying dual on an interface implies that the interface is compatible with Automation, and therefore causes both the TYPEFLAG_FDUAL and TYPEFLAG_FOLEAUTOMATION flags to be set.

Flags

TYPEFLAG_FDUAL, TYPEFLAG_FOLEAUTOMATION

entry(*entryid*)

Description

Identifies the entry point in the DLL.

Allowed on

Functions in a module (required).

Comments

If *entryid* is a string, this is a named entry point. If *entryid* is a number, the entry point is defined by an ordinal. This attribute provides a way to obtain the address of a function in a module.

helpcontext(*numctxt*)

Description

Sets the context in the Help file.

Allowed on

Library, interface, dispinterface, struct, enum, union, module, typedef, method, struct member, enum value, property, coclass, const.

Comments

Retrieved by the **GetDocumentation** functions in the **ITypeLib** and **ITypeInfo** interfaces. The *numctxt* is a 32-bit Help context ID in the Help file.

helpfile(*filename*)

Description

Sets the name of the Help file.

Allowed on

Library.

Comments

Retrieved through the **GetDocumentation** functions in the **ITypeLib** and **ITypeInfo** interfaces.

All types in a library share the same Help file.

helpstring(*string*)

Description

Sets the Help string.

Allowed on

Library, interface, dispinterface, struct, enum, union, module, typedef, method, struct member, enum value, property, coclass, const.

Comments

Retrieved through the **GetDocumentation** functions in the **ITypeLib** and **ITypeInfo** interfaces.

helpstringcontext(*contextid*)

Description

Sets the string context in the Help file.

Allowed on

Type library, type information (TypeInfo), function, and variable level.

Comments

Retrieved by the **GetDocumentation2** functions in the **ITypeLib2** and **ITypeInfo2** interfaces. The *contextid* is a 32-bit Help context ID in the Help file.

helpstringdll(*dllname*)

Description

Sets the name of the DLL to use to perform the doc string lookup (localization).

Allowed on

Type lbrary.

Comments

Retrieved through the **GetDocumentation2** functions in the **ITypeLib2** and **ITypeInfo2** interfaces.

hidden

Description

Indicates that the item exists, but should not be displayed in a user-oriented browser.

Allowed on

Property, method, coclass, dispinterface, interface, library.

Comments

This attribute allows members to be removed from an interface by shielding them from further use, while maintaining compatibility with existing code.

When specified for a library, the attribute prevents the entire library from being displayed. It is intended for use by controls. Hosts need to create a new type library that wraps the control with extended properties.

Flags

VARFLAG_FHIDDEN, FUNCFLAG_FHIDDEN, TYPEFLAG_FHIDDEN

id(*num*)

Description

Identifies the DISPID of the member.

Allowed on

Method or property in an interface or dispinterface.

Comments

The *num* is a 32-bit integral value in the following format:

Bits	Value
0 - 15	Offset. Any value is permissible.
16 - 21	The nesting level of this TypeInfo in the inheritance heirarchy. For example: `interface mydisp : IDispatch` The nesting level of **IUnknown** is 0, **IDispatch** is 1, and MyDisp is 2.
22 - 25	Reserved. Must be zero.
26 - 28	DISPID value.
29	**True** if this is the member ID for a FuncDesc; otherwise **False**.
30-31	Must be 01.

Negative IDs are reserved for use by Automation.

immediatebind

Description

Allows individual bindable properties on a form to specify this behavior. When this bit is set, all changes will be notified.

Flags

VARFLAG_FIMMEDIATEBIND
FUNCFLAG_FIMMEDIATEBINDComments

Allows controls to differentiate two different types of bindable properties. One type of bindable property needs to notify every change to the database (for example, with a check box control where every change needs to be sent through to the underlying database, even though the control has not lost the focus). However, controls such as a list box need to have the change of a property communicated to the database when the control loses focus, because the user may have changed the selection with the arrow keys before finding the desired setting. If the change notification was sent to the database every time the user pressed an arrow key, it would give an unacceptable performance.

The **bindable** and **requestedit** attribute bits need to be set for this new bit to have an effect.

in

Description

Specifies an input parameter.

Allowed on

Parameter.

Comments

The parameter can be a pointer (such as *char**) but the value it refers to is not returned.

lcid

Description

Indicates that the parameter is a locale ID.

Allowed on

Parameter in a member of an interface.

Comments

Only one parameter can have this attribute. The parameter must have the *in* attribute and not the **out** attribute, and its type must be **long**. The **lcid** attribute is not allowed on dispinterfaces.

The **lcid** attribute allows members in th e VTBL to receive an LCID at the time of invocation. By convention, the **lcid** parameter is the last parameter not to have the **retval** attribute. If the member specifies *propertyput* or *propertyputref*, the **lcis** parameter must precede the parameter that represents the right side of the property assignment.

ITypeInfo::Invoke passes the LCID of the type information into the **lcid** parameter. Parameters with this attibute are not displayed in user-oriented browsers.

lcid(*numid*)

Description

This attribute identifies the locale for a type library.

Allowed on

Library.

Comments

The *numid* is a 32-bit locale ID, as used in Win32 National Language Support. The locale ID is typically entered in hexadecimal format.

licensed

Description

Indicates that the class is licensed.

Allowed on

Coclass.

Flags

TYPEFLAG_FLICENSED

nonbrowsable

Description

Indicates that the property appears in an object browser (which does not show property values), but does not appear in a properties browser (which does show propery values).

Allowed on

Property.

Flags

VARFLAG_FNONBROWSABLE, FUNCFLAG_FNONBROWSABLE

noncreatable

Description

Indicates that the class does not support creation at the top level (for example, through **ITypeInfo::CreateInstance** or **CoCreateInstance**). An object of such a class is usually obtained through a method call on another object.

Allowed on

Coclass.

Flags

TYPEFLAG_FCANCREATE

Example

```
[
     uuid(1e196b20-1fc3-1069-996b-00dd010ef671),
     helpstring("This is Dynaset"),
   noncreatable
]
coclass Dynaset
{
     [default] interface IDynaset;
     [default, source] interface IDynasetEvents;
}
```

nonextensible

Description

Indicates that the **IDispatch** implementation includes only the properties and methods listed in the interface description.

Allowed on

Dispinterface, interface.

Comments

The interface must have the **dual** attribute.

By default, Automation assumes that interfaces can add members at run time, meaning that it assumes the interfaces are extensible.

Flags

TYPEFLAG_FNONEXTENSIBLE

odl

Description

Identifies an interface as an Object Description Language (ODL) interface.

Allowed on

Interface (required).

Comments

This attribute must appear on all interfaces.

oleautomation

Description

The **oleautomation** attribute indicates that an interface is compatible with Automation.

Allowed on

Interface.

Comments

Not allowed on dispinterfaces.

The parameters and return types specified for its members must be compatible with Automation, as listed in the following table:

Type	Description
boolean	Data item that can have the value **True** or **False**. The size corresponds to VARIANT_BOOL. Use VT_TRUE, VT_FALSE.
unsigned char	8-bit unsigned data item.
double	64-bit IEEE floating-point number.
float	32-bit IEEE floating-point number.
int	Signed integer, whose size is system dependent.
long	32-bit signed integer.
short	16-bit signed integer.
BSTR	Length-prefixed string, as described in Chapter 5, "Dispatch Interface and API Functions."
CURRENCY	8-byte, fixed-point number.
DATE	64-bit, floating-point fractional number of days since December 30, 1899.
SCODE	Built-in error type that corresponds to VT_ERROR.
typedef enum *myenum*	Signed integer, whose size is system dependent.
interface IDispatch *	Pointer to the **IDispatch** interface (VT_DISPATCH).
interface IUnknown *	Pointer to an interface that does not derive from **IDispatch** (VT_UNKNOWN). (Any OLE interface can be represented by its **IUnknown** interface.)
dispinterface *Typename* *	Pointer to an interface derived from **IDispatch** (VT_DISPATCH).
coclass *Typename* *	Pointer to a coclass name (VT_UNKNOWN).
[**oleautomation**] **interface** *Typename* *	Pointer to an interface that derives from **IDispatch**.
SAFEARRAY(*TypeName***)**	*TypeName* is any of the above types. Array of these types.
TypeName*	*TypeName* is any of the above types. Pointer

to a type.

A parameter is compatible with Automation if its type is compatible with an Automation type, a pointer to an Automation type, or a SAFEARRAY of an Automation type.

A return type is compatible with Automation if its type is an HRESULT or is **void**. Methods in Automation must return either HRESULT or **void**.

A member is compatible with Automation if its return type and all of its parameters are compatible with Automation.

An interface is compatible with Automation if it derives from **IDispatch** or **IUnknown,** if it has the **oleautomation** attribute, or if all of its VTBL entries are compatible with Automation. For 32-bit systems, the calling convention for all methods in the interface must be STDCALL. For 16-bit systems, all methods must have the CDECL calling convention. Every dispinterface is compatible with Automation.

Flags

TYPEFLAG_FOLEAUTOMATION

optional

Description

Specifies an optional parameter.

Allowed on

Parameter.

Comments

Valid only if the parameter is of type VARIANT or VARIANT*. All subsequent parameters of the function must also be **optional**.

out

Description

Specifies an output parameter.

Allowed on

Parameter.

Comments

The parameter must be a pointer to memory that will receive a result.

propget

Description

Specifies a property-accessor function.

Allowed on

Functions, methods in interfaces, dispinterfaces.

Comments

The property must have the same name as the function. At most, one of **propget, propput,** and **propputref** can be specified for a function.

Flags

INVOKE_PROPERTYGET

propput

Description

Specifies a property-setting function.

Allowed on

Functions, methods in interfaces, dispinterfaces.

Comments

The property must have the same name as the function. Only one **propget**, **propput,** and **propputref** can be specified.

Flags

INVOKE_PROPERTYPUT

propputref

Description

Specifies a property-setting function that uses a reference instead of a value.

Allowed on

Functions, methods in interfaces, dispinterfaces.

Comments

The property must have the same name as the function. Only one **propget**, **propput**, and **propputref** can be specified.

Flags

INVOKE_PROPERTYPUTREF

public

Description

Includes an alias declared with the **typedef** keyword in the type library.

Allowed on

Alias declared with **typedef.**

Comments

By default, an alias that is declared with **typedef,** and has no other attributes, is treated as a **#define**, and is not included in the type library. Using the **public** attribute ensures that the alias becomes part of the type library.

readonly

Description

Prohibits assignment to a variable.

Allowed on

Variable.

Flags

VARFLAG_FREADONLY

replaceable

Description

Tags an interface as having default behaviors.

Allowed on

Coclass, variable.

Comments

The object supports **IConnectionPointWithDefault**.

Flags

TYPEFLAG_FREPLACEABLE, FUNCFLAG_FREPLACEABLE, VARFLAG_FREPLACEABLE

requestedit

Description

Indicates that the property supports the **OnRequestEdit** notification.

Allowed on

Property.

Comments

The property supports the **OnRequestEdit** notification, raised by a property before it is edited. An object can support data binding and not have this attribute.

Flags

FUNCFLAG_FREQUESTEDIT, VARFLAG_FREQUESTEDIT

restricted

Description

Prevents the item from being used by a macro programmer.

Allowed on

Type library, type information, coclass member, or member of a module or interface.

Comments

This attribute is allowed on a member of a coclass, independent of whether the member is a dispinterface or interface, and independent of whether the member is a sink or source. A member of a coclass cannot have both the **restricted** and **default** attributes.

Flags

IMPLTYPEFLAG_FRESTRICTED, FUNCFLAG_FRESTRICTED

```
[    odl,
           dual,
           uuid(1e196b20-1f3c-1069-996b-00dd010ef676),
           helpstring("This is IForm"),
           restricted
     ]
     interface IForm: IDispatch
     {
           [propget]
           HRESULT Backcolor([out, retval] long *Value);

           [propput]
           HRESULT Backcolor([in] long Value);
     }


     [     odl,
           dual,
           uuid(1e196b20-1f3c-1069-996b-00dd010ef767),
           helpstring("This is IFormEVents"),
           restricted
     ]
     interface IFormEvents: IDispatch
     {
           HRESULT Click();
     }


     [     uuid(1e196b20-1f3c-1069-996b-00dd010fe676),
           helpstring("This is Form")
     ]
     coclass Form
     {
           [default] interface IForm;
           [default, source] interface IFormEvents;
     }
```

retval

Description

Designates the parameter that receives the return value of the member.

Allowed on

Parameters of interface members that describe methods or get properties.

Comments

This attribute can be used only on the last parameter of the member. The parameter must have the **out** attribute and must be a pointer type.

Parameters with this attribute are not displayed in user-oriented browsers.

Flags

IDLFLAG_FRETVAL

source

Description

Indicates that a member is a source of events.

Allowed on

Member of a coclass, property, or method.

Comments

For a member of a coclass, this attribute indicates that the member is called rather than implemented.

On a property or method, this attribute indicates that the member returns an object or VARIANT that is a source of events. The object implements the interface **IConnectionPointContainer**.

Flags

IMPLTYPEFLAG_FSOURCE, VARFLAG_SOURCE, FUNCFLAG_SOURCE

string

Description

Specifies a string.

Allowed on

Structure, member, parameter, property.

Comments

Included only for compatibility with the Interface Definition Language (IDL). Use LPSTR for a zero-terminated string.

uidefault

Description

Indicates that the type information member is the default member for display in the user interface.

Allowed on

A member of an interface or dispinterface.

Comments

This attribute is used to mark an event as the default (the first one created) or a property as the default (the one to select first in the properties browser).

For example, Visual Basic uses this attribute in the following ways:

- When an object is double-clicked at design time, Visual Basic jumps to the event in the default source interface that is marked as [**uidefault**]. If there is no such member, then Visual Basic displays the first one listed in the default source interface.
- When an object is selected at design time, by default, the Properties window in Visual Basic displays the property in the default interface that is marked as [**uidefault**]. If there is no such member, then Visual Basic displays the first one listed in the default interface.

Flags

FUNCFLAG_FUIDEFAULT, VARFLAG_FUIDEFAULT

```
  [   odl,
      dual,
      uuid(1e196b20-1f3c-1069-996b-00dd010ef676),
      restricted
]
interface IForm: IDispatch
{
      [propget]
      HRESULT Backcolor([out, retval] long *Value);

      [propput]
      HRESULT Backcolor([in] long Value);


      [propget, uidefault]
      HRESULT Name([out, retval] BSTR *Value);

      [propput, uidefault]
      HRESULT Name([in] BSTR Value);
}

[     odl,
      dual,
      uuid(1e196b20-1f3c-1069-996b-00dd010ef767),
      restricted
]
interface IFormEvents: IDispatch
{
    [uidefault]
      HRESULT Click();
```

```
     HRESULT Resize();
}

[uuid(1e196b20-1f3c-1069-996b-00dd010fe676)]
coclass Form
{
     [default] interface IForm;
     [default, source] interface IFormEvents;
}
```

uuid(*uuidval*)

Description

Specifies the universally unique ID (UUID) of the item.

Allowed on

Required for library, dispinterface, interface, and coclass. Optional for struct, enum, union, module, and typedef.

Comments

The **uuidval** is a 16-byte value using hexadecimal digits in the following format: `12345678-1234-1234-1234-123456789ABC`. This value is returned in the **TypeAttr** structure retrieved by **TypeInfo::GetTypeAttr.**

vararg

Description

Indicates a variable number of arguments.

Allowed on

Function.

Comments

Indicates that the last parameter is a safe array of VARIANT type, which contains all of the remaining parameters.

version(*versionval*)

Description

Specifies a version number.

Allowed on

Library, struct, module, dispinterface, interface, coclass, enum, union.

Comments

The argument *versionval* is a real number in the format *n.m*, where *n* is a major version number and *m* is a minor version number.

ODL Statements and Directives

The following sections describe the statements and directives that make up the Object Description Language (ODL).

coclass Statement

Describes the globally unique ID (GUID) and the supported interfaces for a Component Object Model (COM).

Syntax

[*attributes***]**
coclass *classname* **{**
[*attributes2*] [**interface** | **dispinterface**] *interfacename*;
// Code omitted here for brevity.
};

Syntax Elements

attributes
The **uuid** attribute is required on a coclass. This is the same **uuid** that is registered as a CLSID in the system registration database. The **helpstring, helpcontext, licensed, version, control, hidden**, and **appobject** attributes are accepted, but not required, before a coclass definition. For more information about the attributes accepted before a coclass definition, see "Attribute Descriptions" earlier in this chapter. The **appobject** attribute makes the functions and properties of the coclass globally available in the type library.

classname
Name by which the common object is known in the type library.

attributes2
Optional attributes for the interface or dispinterface. The **source, default,** and **restricted** attributes are accepted on an interface or dispinterface in a coclass.

interfacename
Either an interface declared with the **interface** keyword, or a dispinterface declared with the **dispinterface** keyword.

Comments

The Component Object Model defines a class as an implementation that allows **QueryInterface** between a set of interfaces.

Example

```
[ uuid(BFB73347-822A-1068-8849-00DD011087E8), version(1.0), helpstring("A
class"), helpcontext(2481), appobject]
coclass myapp {
     [source] interface IMydocfuncs;
     dispinterface DMydocfuncs;
};

[uuid 00000000-0000-0000-0000-123456789019]
coclass foo
{
     [restricted] interface bar;
     interface bar;
}
```

dispinterface Statement

Defines a set of properties and methods on which **IDispatch::Invoke** can be called. A dispinterface can be defined by explicitly listing the set of supported methods and properties (Syntax 1) or by listing a single interface (Syntax 2).

Syntax 1

[*attributes***]**
dispinterface *intfname* **{**
properties:
proplist
methods:
methlist
};

Syntax 2

[*attributes***]**
dispinterface *intfname* **{**
interface *interfacename*
};

Syntax Elements

attributes
The **helpstring, helpcontext, hidden, uuid,** and **version** attributes are accepted before **dispinterface**. For more information about the attributes accepted before a **dispinterface** definition, see "Attribute Descriptions" earlier in this chapter. Attributes (including the brackets) can be omitted, except for the **uuid** attribute, which is required.

intfname
The name by which the dispinterface is known in the type library. This name must be unique within the type library.

interfacename
(Syntax 2) The name of the interface to declare as an **IDispatch** interface.

proplist
(Syntax 1) An optional list of properties supported by the object, declared in the form of variables. This is the short form for declaring the property functions in the methods list. See the comments section for details.

methlist
(Syntax 1) A list comprising a function prototype for each method and property in the dispinterface. Any number of function definitions can appear in *methlist*. A function in *methlist* has the following form:

[*attributes***]** *returntype methname***(***params***);**

The following attributes are accepted on a method in a dispinterface: **helpstring, helpcontext, string** (for compatibility with the Interface Definition Language), **bindable, defaultbind, displaybind, propget, propput, propputref,** and **vararg**. If **vararg** is specified, the last parameter must be a safe array of VARIANT type.

The parameter list is a comma-delimited list, each element of which has the following form:

[*attributes***]** *type paramname*

The *type* can be any declared or built-in type, or a pointer to any type. Attributes on parameters are **in, out, optional,** and **string**.

If **optional** is specified, it must only be specified on the right-most parameters, and the types of those

parameters must be VARIANT.

Comments

Method functions are specified exactly as described in the "module Statement," except that the **entry** attribute is not allowed.

Note Stdole32.tlb (Stdole.tlb on 16-bit systems) must be imported, because a dispinterface inherits from **IDispatch**.

Properties can be declared either in the properties or methods lists. Declaring properties in the properties list does not indicate the type of access the property supports (**get, put,** or **putref**). Specify the **readonly** attribute for properties that do not support **put** or **putref**. If the property functions are declared in the methods list, functions for one property will all have the same ID.

Using Syntax 1, the *properties:* and *methods:* tags are required. The **id** attribute is also required on each member. For example:

```
properties:
     [id(0)] int Value;      // Default property.
methods:
     [id(1)] void Show();
```

Unlike interface members, dispinterface members cannot use the **retval** attribute to return a value in addition to an HRESULT error code. The **lcid** attribute is also invalid for dispinterfaces because **IDispatch::Invoke** passes a locale ID (LCID). However, it is possible to declare an interface again that uses these attributes.

Using Syntax 2, interfaces that support **IDispatch** and are declared earlier in an Object Definition Language (ODL) script can be redeclared as **IDispatch** interfaces as follows:

```
dispinterface helloPro {
     interface hello;
};
```

This example declares all of the members of the Hello sample and all of the members that it inherits to support **IDispatch**. In this case, if Hello was declared earlier with **lcid** and **retval** members that returned HRESULTs, MkTypLib would remove each **lcid** parameter and HRESULT return type, and instead mark the return type as that of the **retval** parameter.

The properties and methods of a dispinterface are not part of the VTBL of the dispinterface. Consequently, **CreateStdDispatch** and **DispInvoke** cannot be used to implement **IDispatch::Invoke**. The dispinterface is used when an application needs to expose existing non-VTBL functions through Automation. These applications can implement **IDispatch::Invoke** by examining the *dispidMember* parameter and directly calling the corresponding function.

Example

```
[uuid(BFB73347-822A-1068-8849-00DD011087E8), version(1.0),
helpstring("Useful help string."), helpcontext(2480)]
dispinterface MyDispatchObject {
     properties:
          [id(1)] int x;    // An integer property named x.
          [id(2)] BSTR y;   // A string property named y.
```

```
     methods:
           [id(3)] void show();          // No arguments, no result.
           [id(11)] int computeit(int inarg, double *outarg);
};

[uuid 00000000-0000-0000-0000-123456789012]
dispinterface MyObject
{
     properties:
     methods:
           [id(1), propget, bindable, defaultbind, displaybind]
           long x();

           [id(1), propput, bindable, defaultbind, displaybind]
           void x(long rhs);
}
```

enum Statement

Defines a C-style enumerated type.

Syntax

typedef [*attributes***] enum** [*tag*] **{**
    enumlist
} *enumname***;**

Syntax Elements

attributes
The **helpstring, helpcontext, hidden,** and **uuid** attributes are accepted before an **enum** statement. The **helpstring** and **helpcontext** attributes are accepted on an enumeration element. For more information about the attributes accepted before an enumeration definition, see "Attribute Descriptions" earlier in this chapter. Attributes (including the brackets) can be omitted. If *uuid* is omitted, the enumeration is not uniquely specified in the system.

tag
An optional tag, as with a C **enum**.

enumlist
List of enumerated elements.

enumname
Name by which the enumeration is known in the type library.

Comments

The **enum** keyword must be preceded by **typedef**. The enumeration description must precede other references to the enumeration in the library. If **value** is not specified for enumerators, the numbering progresses, as with enumerations in C. The type of the **enum** element is **int,** the system default integer, which depends on the target type library specification.

Examples

```
typedef [uuid(DEADF00D-C0DE-B1FF-F001-A100FF001ED),
           helpstring("Farm Animals are friendly"), helpcontext(234)]
enum {
     [helpstring("Moo")] cows = 1,
     pigs = 2
} ANIMALS;
```

importlib Directive

Makes types that have already been compiled into another type library available to the library currently being created. All **importlib** directives must precede the other type descriptions in the library.

Syntax

importlib(*filename*);

Syntax Element

filename

The location of the type library file when MkTypLib is executed.

Comments

The **importlib** directive makes any type defined in the imported library accessible from within the library being compiled. Ambiguity is resolved as the current library is searched for the type. If the type cannot be found, MkTypLib searches the imported library that is lexically first, and then the next, and so on. To import a type name in code, the name should be entered as *libname.typename,* where *libname* is the library name as it appeared in the **library** statement when the library was compiled.

The imported type library should be distributed with the library being compiled.

Example

The following example imports the libraries Stdole.tlb and Mydisp.tlb:

```
library BrowseHelper
{
     importlib("stdole.tlb");
     importlib("mydisp.tlb");
// Additional text omitted.
}
```

interface Statement

Defines an interface, which is a set of function definitions. An interface can inherit from any base interface.

Syntax

[*attributes***]**
interface *interfacename* [**:***baseinterface*] **{**
functionlist
};

Syntax Elements

attributes

The attributes **dual**, **helpstring**, **helpcontext**, **hidden**, **odl**, **oleautomation**, **uuid**, and **version** are accepted before **interface**. If the interface is a member of a coclass, the attributes **source**, **default**, and **restricted** are also accepted. For more information about the attributes that can be accepted before an **interface** definition, refer to the section "Attribute Descriptions" earlier in this chapter.

The attributes **odl** and **uuid** are required on all **interface** declarations.

interfacename

The name by which the interface is known in the type library.

baseinterface

The name of the interface that is the base class for this interface.

functionlist

List of function prototypes for each function in the interface. Any number of function definitions can appear in the function list. A function in the function list has the following form:

[*attributes***]** *returntype* **[***calling convention***]** *funcname***(***params***);**

The following attributes are accepted on a function in an interface: **helpstring**, **helpcontext**, **string**, **propget**, **propput**, **propputref**, **bindable**, **defaultbind**, **displaybind**, and **vararg**. If **vararg** is specified, the last parameter must be a safe array of VARIANT type. The optional calling convention can be **__pascal**/**_pascal**/**pascal**, **__cdecl**/**_cdecl**/**cdecl**, or **__stdcall**/**_stdcall**/**stdcall**. The calling convention specification can include up to two leading underscores.

The parameter list is a comma-delimited list, as follows:

[*attributes***]** *type paramname*

The *type* can be any previously declared type, built-in type, a pointer to any type, or a pointer to a built-in type. Attributes on parameters are **in**, **out**, **optional**, and **string**.

If **optional** is used, it must be specified only on the right-most parameters, and the types of those parameters must be VARIANT.

Comments

Because the functions described by the **interface** statement are in the VTBL, **DispInvoke** and **CreateStdDispatch** can be used to provide an implementation of **IDispatch::Invoke**. For this reason, **interface** is more commonly used than **dispinterface** to describe the properties and methods of an object.

Functions in interfaces are the same as described in "The module Statement," except that the **entry** attribute is not allowed.

Members of interfaces that need to raise exceptions should return an HRESULT and specify a **retval** parameter for the actual return value. The **retval** parameter is always the last parameter in the list.

Examples

The following example defines an interface named Hello with two member functions, **HelloProc** and **Shutdown**:

```
[uuid(BFB73347-822A-1068-8849-00DD011087E8), version(1.0)]
interface hello : IUnknown
{
void HelloProc([in, string] unsigned char * pszString);
void Shutdown(void);
};
```

The next example defines a dual interface named **IMyInt**, which has a pair of accessor functions for the **MyMessage** property, and a method that returns a string.

```
[dual]
interface IMyInt : IDispatch
{
     // A property that is a string.
     [propget] HRESULT MyMessage([in, lcid] LCID lcid,
                                                [out, retval] BSTR
*pbstrRetVal);
     [propput] HRESULT MyMessage([in] BSTR rhs, [in, lcid] DWORD lcid);

     // A method returning a string.
     HRESULT SayMessage([in] long NumTimes,
                              [in, lcid] DWORD lcid,
                              [out, retval] BSTR *pbstrRetVal);
}
```

The members of this interface return error information and function return values through the HRESULT values and **retval** parameters, respectively. Tools that access the members can return the HRESULT to their users, or can simply expose the **retval** parameter as the return value, and handle the HRESULT transparently.

A dual interface must derive from **IDispatch**.

library Statement

Describes a type library. This description contains all of the information in a MkTypLib input file (ODL).

Syntax

[*attributes***] library** *libname* **{**
    definitions
};

Syntax Elements

attributes
The **helpstring, helpcontext, lcid, restricted, hidden, control,** and **uuid** attributes are accepted before a **library** statement. For more information about the attributes accepted before a **library** definition, see "Attribute Descriptions" earlier in this chapter. The **uuid** attribute is required.

libname
The name by which the type library is known.

definitions
Descriptions of any imported libraries, data types, modules, interfaces, dispinterfaces, and coclasses relevant to the object being exposed.

Comments

The **library** statement must precede any other type definitions.

Example

```
[
      uuid(F37C8060-4AD5-101B-B826-00DD01103DE1), // LIBID_Hello.
      helpstring("Hello 2.0 Type Library"),
      lcid(0x0409),
      version(2.0)
]
library Hello
{
      importlib("stdole.tlb");
      [
            uuid(F37C8062-4AD5-101B-B826-00DD01103DE1),     // IID_Ihello.
            helpstring("Application object for the Hello application."),
            oleautomation,
            dual
      ]
      interface IHello : IDispatch
      {
            [propget, helpstring("Returns the application of the object.")]
            HRESULT Application([in, lcid] long localeID,
                  [out, retval] IHello** retval)
      }
}
```

module Statement

Defines a group of functions, typically a set of DLL entry points.

Syntax

[*attributes***]**
module *modulename* **{**
elementlist
};

Syntax Elements

attributes

The attributes **uuid, version, helpstring, helpcontext, hidden,** and **dllname** are accepted before a **module** statement. For more information about the attributes that can be accepted before a module definition, see "Attribute Descriptions" earlier in this chapter. The **dllname** attribute is required. If **uuid** is omitted, the module is not uniquely specified in the system.

modulename

The name of the module.

elementlist

List of constant definitions and function prototypes for each function in the DLL. Any number of function definitions can appear in the function list. A function in the function list has the following form:

[*attributes***]** *returntype* **[***calling convention***]** *funcname***(***params***);**
[*attributes***]** **const** *constname* **=** *constval***;**

Only the attributes **helpstring** and **helpcontext** are accepted for a **const**.

The following attributes are accepted on a function in a module: **helpstring, helpcontext, string, entry, propget, propput, propputref, vararg.** If **vararg** is specified, the last parameter must be a safe array of VARIANT type.

The optional *calling convention* can be one of **__pascal/_pascal/pascal**, **__cdecl/_cdecl/cdecl**, or **__stdcall/_stdcall/stdcall**. The calling convention can include up to two leading underscores.

The parameter list is a comma-delimited list.

[*attributes***]** *type paramname*

The *type* can be any previously declared type or built-in type, a pointer to any type, or a pointer to a built-in type. Attributes on parameters are **in, out,** and **optional**.

If **optional** is specified, it must only be specified on the right-most parameters, and the types of those parameters must be VARIANT.

Comments

The header file (.h) output for modules is a series of function prototypes. The **module** keyword and surrounding brackets are stripped from the header file output, but a comment (\\ **module** *modulename*) is inserted before the prototypes. The keyword **extern** is inserted before the declarations.

Example

```
[uuid(D00BED00-CEDE-B1FF-F001-A100FF001ED),
     helpstring("This is not GDI.EXE"), helpcontext(190),
     dllname("MATH.DLL")]
module somemodule{
     [helpstring("Color for the frame")] unsigned long const COLOR_FRAME
          = 0xH80000006;
```

```
    [helpstring("Not a rectangle but a square"), entry(1)] pascal double
square([in] double x);
};
```

struct Statement

Defines a C-style structure.

Syntax

typedef [*attributes***]**
struct [*tag*] **{**
    memberlist
} *structname***;**

Syntax Elements

attributes
The attributes **helpstring, helpcontext, uuid, hidden**, and **version** are accepted before a **struct** statement. The attributes **helpstring, helpcontext,** and **string** are accepted on a structure member. For more information about the attributes accepted before a structure definition, see "Attribute Descriptions" earlier in this chapter. Attributes (including the brackets) can be omitted. If **uuid** is omitted, the structure is not specified uniquely in the system.

tag
An optional tag, as with a C **struct**.

memberlist
List of structure members defined with C syntax.

structname
Name by which the structure is known in the type library.

Comments

The **struct** keyword must be preceded with a **typedef**. The structure description must precede other references to the structure in the library. Members of a **struct** can be of any built-in type, or any type defined lexically as a **typedef** before the **struct**. For a description of how strings and arrays can be entered, see the sections "String Definitions" and "Array Definitions" earlier in this chapter.

Example

```
typedef [uuid(BFB7334B-822A-1068-8849-00DD011087E8),
     helpstring("A task"), helpcontext(1019)]
struct {
     DATE startdate;
     DATE enddate;
     BSTR ownername;
     SAFEARRAY (int) subtasks;
     int A_C_array[10];
} TASKS;
```

typedef Statement

Creates an alias for a type.

Syntax

typedef [*attributes***]** *basetype aliasname***;**

Syntax Elements

attributes

Any attribute specifications must follow the **typedef** keyword. If no attributes and no other type (for example, **enum, struct**, or **union**) are specified, the alias is treated as a **#define** and does not appear in the type library. If no other attribute is desired, **public** can be used to explicitly include the alias in the type library. The **helpstring, helpcontext**, and **uuid** attributes are accepted before a **typedef**. For more information, see "Attribute Descriptions" earlier in this chapter. If **uuid** is omitted, the **typedef** is not uniquely specified in the system.

basetype

The type for which the alias is defined.

aliasname

Name by which the type will be known in the type library.

Comments

The **typedef** keyword must also be used whenever a **struct** or **enum** is defined. The name recorded for the **enum** or **struct** is the **typedef** name, and not the tag for the enumeration. No attributes are required to make sure the alias appears in the type library.

Enumerations, structures, and unions must be defined with the **typedef** keyword. The attributes for a type defined with **typedef** are enclosed in brackets following the **typedef** keyword. If a simple alias **typedef** has no attributes, it is treated like a **#define**, and the *aliasname* does not appear in the library. Any attribute (**public** can be used if no others are desired) specified between the **typedef** keyword and the rest of a simple alias definition causes the alias to appear explicitly in the type library. The attributes typically include such items as a Help string and Help context.

Examples

```
typedef [public]  long DWORD;
```

This example creates a type description for an alias type with the name DWORD.

```
typedef enum {
            TYPE_FUNCTION = 0,
            TYPE_PROPERTY = 1,
            TYPE_CONSTANT = 2,
            TYPE_PARAMETER = 3
      } OBJTYPE;
```

The second example creates a type description for an enumeration named OBJTYPE, which has four enumerator values.

union Statement

Defines a C-style union.

Syntax

typedef [*attributes***] union** [*tag*] **{**
    memberlist
} *unionname*;

Syntax Elements

attributes
: The attributes **helpstring, helpcontext, uuid,** **hidden,** and **version** are accepted before a **union**. The **helpstring, helpcontext,** and **string** attributes are accepted on a **union** member. For more information about the attributes accepted before a union definition, see "Attribute Descriptions" earlier in this chapter. Attributes (including the square brackets) can be omitted. If **uuid** is omitted, the union is not uniquely specified in the system.

tag
: An optional tag, as with a C **union**.

memberlist
: List of **union** members defined with C syntax.

unionname
: Name by which the **union** is known in the type library.

Comments

The **union** keyword must be preceded with a **typedef**. The **union** description must precede other references to the structure in the library. Members of a **union** can be of any built-in type, or any type defined lexically as a **typedef** before the **union**. For a description of how strings and arrays can be entered, see the sections "String Definitions" and "Array Definitions" earlier in this chapter.

Example

```
[uuid(BFB7334C-822A-1068-8849-00DD011087E8), helpstring("A task"),
helpcontext(1019)]
typedef union {
     COLOR polycolor;
     int   cVertices;
     boolean filled;
     SAFEARRAY (int) subtasks;
} UNIONSHOP;
```

Type Description Interfaces

Type description interfaces provide a way to read and bind to the descriptions of objects in a type library. These descriptions are used by ActiveX clients when they browse, create, and manipulate ActiveX (Automation) objects.

The type description interfaces described in this chapter include:

- **ITypeLib** –Retrieves information about a type library.
- **ITypeInfo** – Reads the type information within the type library.
- **ITypeComp** – Creates compilers that use type information.

This chapter also describes functions for loading, registering, and querying type libraries.

Overview of Type Description Interfaces

A type library is a container for type descriptions of one or more objects, and is accessed through the **ITypeLib** interface. The **ITypeLib** interface provides access to information about the type description in a type library. The descriptions of individual objects are accessed through the **ITypeInfo** interface.

The following table describes the member functions of each of the type description interfaces:

Category	Member function	Purpose
ITypeLib	**FindName**	Finds occurrences of a type description in a type library.
	GetDocumentation	Retrieves the library's documentation string, name of the complete Help file name and path, and the context ID for the library Help topic.
	GetLibAttr	Retrieves the structure containing the library's attributes.
	GetTypeComp	Retrieves a pointer to the **ITypeComp** for a type library. This enables a client compiler to bind to the library's types, variables, constants, and global functions.
	GetTypeInfo	Retrieves the specified type description in the library.
	GetTypeInfoCount	Retrieves the number of type descriptions in the library.
	GetTypeInfoOfGuid	Retrieves the type description corresponding to the specified GUID.
	GetTypeInfoType	Retrieves the type of a type description.
	IsName	Indicates whether a passed-in string contains the name of a type, or a member described in the library.
	ReleaseTLibAttr	Releases TLIBATTR, originally obtained from **ITypeLib::GetLibAttr**.
ITypeInfo	**AddressOfMember**	Retrieves the addresses of static functions or variables, such as those defined in a DLL.
	CreateInstance	Creates a new instance of a type that describes a component object class (coclass).
	GetContainingTypeLib	Retrieves both the type library that contains a specific type description and the index of

	the type description within the type library.
GetDllEntry	Retrieves a description or specification of an entry point for a function in a DLL.
GetDocumentation	Retrieves the documentation string, name of the complete Help file name and path, and the context ID for the Help topic for a specified type description.
GetFuncDesc	Retrieves the FUNCDESC structure that contains information about a specified function.
GetIDsOfNames	Maps between member names and member IDs, and parameter names and parameter IDs.
GetImplTypeFlags	Retrieves the IMPLTYPE flags for an interface.
GetMops	Retrieves marshaling information.
GetNames	Retrieves the variable with the specified member ID, or the name of the function and parameter names corresponding to the specified function ID.
GetRefTypeInfo	Retrieves the type descriptions referenced by a given type description.
GetRefTypeOfImplType	Retrieves the type description of implemented interface types for a coclass or an inherited interface.
GetTypeAttr	Retrieves a TYPEATTR structure that contains the attributes of the type description.
GetTypeComp	Retrieves the **ITypeComp** interface for the type description, which enables a client compiler to bind to the type description's members.
GetVarDesc	Retrieves a VARDESC structure that describes the specified variable.
Invoke	Invokes a method or accesses a property of an object that implements the interface described by the type description.

	ReleaseFuncDesc	Releases a FUNCDESC previously returned by **GetFuncDesc**.
	ReleaseTypeAttr	Releases a TYPEATTR previously returned by **GetTypeAttr**.
	ReleaseVarDesc	Releases a VARDESC previously returned by **GetVarDesc**.
ITypeComp	**Bind**	Maps a name to a member of a type, or binds global variables and functions contained in a type library.
	BindType	Binds to the type descriptions contained within a type library.

ITypeLib Interface

The data that describes a set of objects is stored in a type library. A type library can be a stand-alone binary file (.tlb), a resource in a DLL or executable file, or part of a compound document file.

Implemented by	Used by	Header file name
Oleaut32.dll (32-bit systems) Typelib.dll (16-bit systems)	Tools that need to access the descriptions of objects contained in type libraries.	Oleauto.h Dispatch.h

The system registry contains a list of all the installed type libraries. Type library organization is illustrated in the following figure:

{ewc msdncd, EWGraphic, bsd23529 0 /a "SDK_10.WMF"}

The **ITypeLib** interface provides methods for accessing a library of type descriptions. This interface supports the following:

- Generalized containment for type information. **ITypeLib** allows iteration over the type descriptions contained in the library.
- Global functions and data. A type library can contain descriptions of a set of modules, each of which is the equivalent of a C or C++ source file that exports data and functions. The type library supports compiling references to the exported data and functions.
- General information, including a user-readable name for the library and help for the library as a whole.

ITypeLib::FindName

HRESULT ITypeLib::FindName(

OLECHAR FAR* *szNameBuf*,
unsigned long *lHashVal*,
ITypeInfo FAR* FAR* *rgptinfo*,
MEMBERID FAR* *rgmemid*,
unsigned int FAR* *pcFound*
);

Finds occurrences of a type description in a type library. This may be used to quickly verify that a name exists in a type library.

Parameters

szNameBuf

The name to search for.

lHashVal

A hash value to speed up the search, computed by the **LHashValOfNameSys** function. If *lHashVal* = 0, a value is computed.

rgptinfo

On return, an array of pointers to the type descriptions that contain the name specified in *szNameBuf*. Cannot be Null.

rgmemid

An array of the MEMBERIDs of the found items; *rgmemid*[*i*] is the MEMBERID that indexes into the type description specified by *rgptinfo*[*i*]. Cannot be Null.

pcFound

On entry, indicates how many instances to look for. For example, **pcFound* = 1 can be called to find the first occurrence. The search stops when one is found.

On exit, indicates the number of instances that were found. If the *in* and *out* values of **pcFound* are identical, there may be more type descriptions that contain the name.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.

TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_CANTLOADLIBRARY	The library or .dll file could not be loaded.
TYPE_E_ELEMENTNOTFOUND	The element was not found.

Comments

Passing **pcFound* = *n* indicates that there is enough room in the *rgptinfo* and *rgmemid* arrays for *n* (*ptinfo*, *memid*) pairs. The function returns MEMBERID_NIL in *rgmemid*[*i*], if the name in *szNameBuf* is the name of the type information in *rgptinfo*[*i*].

ITypeLib::GetDocumentation

HRESULT ITypeLib::GetDocumentation(

int *index*,
BSTR FAR* *lpbstrName*,
BSTR FAR* *lpbstrDocString*,
unsigned long FAR* *lpdwHelpContext*,
BSTR FAR* *lpbstrHelpFile*
);

Retrieves the library's documentation string, the complete Help file name and path, and the context ID for the library Help topic in the Help file.

Parameters

index

Index of the type description whose documentation is to be returned. I If *index* is-1, then the documentation for the library itself is returned.

lpbstrName

Returns a BSTR that contains the name of the specified item. If the caller does not need the item name, then *lpbstrName* can be Null.

lpbstrDocString

Returns a BSTR that contains the documentation string for the specified item. If the caller does not need the documentation string, then *lpbstrDocString* can be Null.

lpdwHelpContext

Returns the Help context ID associated with the specified item. If the caller does not need the Help context ID, then *lpdwHelpContext* can be Null.

lpbstrHelpFile

Returns a BSTR that contains the fully qualified name of the Help file. If the caller does not need the Help file name, then *lpbstrHelpFile* can be Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_ELEMENTNOTFOUND	The element was not found.

Comments

The caller should free the BSTR parameters *lpbstrName*, *lpbstrDocString*, and *lpbstrHelpFile*.

Example

```
for (i = 0; i < utypeinfoCount; i++)
{
     CHECKRESULT(ptlib->GetDocumentation(i, &bstrName,
                                                              NULL, NULL,
NULL));
.
.
.
     SysFreeString(bstrName);
}
```

ITypeLib::GetLibAttr

HRESULT ITypeLib::GetLibAttr(

TLIBATTR FAR* FAR* *lplptlibattr*
);

Retrieves the structure that contains the library's attributes.

Parameter

lplptlibattr

Pointer to a structure that contains the library's attributes.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an unsupported format.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

Use **ITypeLib::ReleaseTLibAttr** to free the memory occupied by the TLIBATTR structure.

ITypeLib::GetTypeComp

HRESULT ITypeLib::GetTypeComp(

ITypeComp FAR* FAR* *lplptcomp*
);

Enables a client compiler to bind to a library's types, variables, constants, and global functions.

Parameter

lplptcomp

Points to a pointer to the **ITypeComp** instance for this **ITypeLib.** A client compiler uses the methods in the TypeComp interface to bind to types in **ITypeLib,** as well as to the global functions, variables, and constants defined in **ITypeLib**.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

The **Bind** function of the returned **TypeComp** binds to global functions, variables, constants, enumerated values, and coclass members. The **Bind** function also binds the names of the TYPEKIND enumerations of TKIND_MODULE, TKIND_ENUM, and TKIND_COCLASS. These names shadowany global names defined within the type information. The members of TKIND_ENUM, TKIND_MODULE, and TKIND_COCLASS types marked as Application objects can be directly bound to from **ITypeComp** without specifying the name of the module.

ITypeComp::Bind and **ITypeComp::BindType** accept only unqualified names. **ITypeLib::GetTypeComp** returns a pointer to the **ITypeComp** interface, which is then used to bind to global elements in the library. The names of some types (TKIND_ENUM, TKIND_MODULE, and TKIND_COCLASS) share the name space with variables, functions, constants, and enumerators.

If a member requires qualification to differentiate it from other items in the name space, **GetTypeComp** can be called successively for each qualifier in order to bind to the desired member. This allows programming language compilers to access members of modules, enumerations, and coclasses, even though the member can't be bound to with a qualified name.

ITypeLib::GetTypeInfo

HRESULT ITypeLib::GetTypeInfo(

unsigned int *index*,
ITypeInfo FAR* FAR* *lplpitinfo*
);

Retrieves the specified type description in the library.

Parameters

index

Index of the **ITypeInfo** interface to be returned.

lplpitinfo

If successful, returns a pointer to the pointer to the **ITypeInfo** interface.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
TYPE_E_ELEMENTNOTFOUND	The *index* parameter is outside the range of 0 to **GetTypeInfoCount()** -1.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_REGISTRYACCESS	There was an error accessing the system registration database.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

For dual interfaces, **ITypeLib::GetTypeInfo** returns only the TKIND_DISPATCH type information. To get the TKIND_INTERFACE type information, **ITypeInfo::GetRefTypeOfImplType** can be called on the TKIND_DISPATCH type information, passing an *index* of -1. Then, the returned type information handle can be passed to **ITypeInfo::GetRefTypeInfo**.

Example

The following example gets the TKIND_INTERFACE type information for a dual interface.

```
ptlib->GetTypeInfo((unsigned int) dwIndex, &ptypeinfoDisp);
ptypeinfoDisp->GetRefTypeOfImplType(-1, &phreftype);
ptypeinfoDisp->GetRefTypeInfo(phreftype, &ptypeinfoInt);
```

ITypeLib::GetTypeInfoCount

HRESULT ITypeLib::GetInfoCount(

unsigned int FAR* *pctInfo*
);

Parameter

pctInfo

Points to the location that receives the number of type information interfaces provided by the object.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_NOTIMPL	Failure.

Comments

Returns the number of type descriptions in the type library.

ITypeLib::GetTypeInfoOfGuid

HRESULT ITypeLib::GetTypeInfoOfGuid(

REFGUID *lpguid*,
ITypeInfo FAR* FAR* *lplpitinfo*
);

Retrieves the type description that corresponds to the specified globally unique ID (GUID).

Parameters

lpguid

Pointer to the globally unique ID of the type description.

lplpitinfo

Pointer to a pointer to the **ITypeInfo** interface.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
TYPE_E_ELEMENTNOTFOUND	No type description was found in the library with the specified GUID.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_REGISTRYACCESS	There was an error accessing the system registration database.
TYPE_E_INVALIDSTATE	The type library could not be opened.

ITypeLib::GetTypeInfoType

HRESULT ITypeLib::GetTypeInfoType(*index*, *ptypekind***)**
unsigned int *index*
TYPEKIND FAR* *ptypekind*

Retrieves the type of a type description.

Parameters

index

The index of the type description within the type library.

ptypekind

A pointer to the TYPEKIND enumeration for the type description.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
TYPE_E_ELEMENTNOTFOUND	*Index* is outside the range of 0 to **GetTypeInfoCount()** -1.

ITypeLib::IsName

HRESULT ITypeLib::IsName(

OLECHAR FAR* *szNameBuf*,
unsigned long *lHashVal*,
BOOL *lpfName*
);

Indicates whether a passed-in string contains the name of a type or member described in the library.

Parameter

szNameBuf

The string to test. If **IsName()** is successful, *szNameBuf* is modified to match the case (capitalization) found in the type library.

lHashVal

The hash value of *szNameBuf*.

lpfName

On return, set to True if *szNameBuf* was found in the type library; otherwise False.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.

ITypeLib::ReleaseTLibAttr

HRESULT ITypeLib::ReleaseTLibAttr(

TLIBATTR FAR* *lptlibattr*
);

Releases the TLIBATTR originally obtained from **ITypeLib::GetLibAttr**.

Parameter

lptlibattr

Pointer to the TLIBATTR to be freed.

Comments

Releases the specified TLIBATTR. This TLIBATTR was previously obtained with a call to **GetTypeLib::GetLibAttr**.

ITypeInfo Interface

This section describes **ITypeInfo**, an interface typically used for reading information about objects. For example, an object browser tool can use **ITypeInfo** to extract information about the characteristics and capabilities of objects from type libraries.

Implemented by	Used by	Header file name
Oleaut32.dll (32-bit systems) Typelib.dll (16-bit systems)	Tools that need to access the descriptions of objects contained in type libraries.	Oleauto.h Dispatch.h

Type information interfaces are intended to describe the parts of the application that can be called by outside clients, rather than those that might be used internally to build an application.

The **ITypeInfo** interface provides access to the following:

- The set of function descriptions associated with the type. For interfaces, this contains the set of member functions in the interface.
- The set of data member descriptions associated with the type. For structures, this contains the set of fields of the type.
- The general attributes of the type, such as whether it describes a structure, an interface, and so on.

The type description of an **IDispatch** interface can be used to implement the interface. For more information, see the description of **CreateStdDispatch** in Chapter 5, "Dispatch Interface and API Functions."

An instance of **ITypeInfo** provides various information about the type of an object, and is used in different ways. A compiler can use an **ITypeInfo** to compile references to members of the type. A type interface browser can use it to find information about each member of the type. An **IDispatch** implementor can use it to provide automatic delegation of **IDispatch** calls to an interface.

Type Descriptions

The information associated with an object described by **ITypeInfo** can include a set of functions, a set of data members, and various type attributes. It is essentially the same as the information described by a C++ class declaration, which can be used to define both interfaces and structures, as well as any combination of functions and data members. In addition to interfaces and structure definitions, the **ITypeInfo** interface is used to describe other types, including enumerations and aliases. Because the interface to a C file or library is simply a set of functions and variable declarations, **ITypeInfo** can also be used to describe them.

Type information comprises individual type descriptions. Each type description must have one of the following forms:

Category	ODL keyword	Description
alias	**typedef**	An alias for another type.
enumeration	**enum**	An enumeration.
structure	**struct**	A structure.
union	**union**	A single data item that can have one of a specified group of types.
module	**module**	Data and functions not accessed through VTBL entries.
IDispatch interface	**dispinterface**	**IDispatch** properties and methods accessed through **IDispatch::Invoke**.
OLE interface	**interface**	OLE member functions accessed through VTBL entries.
dual interface	**dual**	Supports either VTBL or **IDispatch**.
component object class	**coclass**	A component object class. Specifies an implementation of one or more OLE interfaces and one or more **IDispatch** interfaces.

Note All bit flags that are not used specifically should be set to zero for future compatibility.

Alias

An alias has TypeKind = TKIND_ALIAS. An alias is an empty set of functions, an empty set of data members, and a type description (located in the TYPEATTR), which gives the actual type definition (**typedef**) of the alias.

Enumeration

An enumeration (**enum**) has TypeKind = TKIND_ENUM. An enumeration is an empty set of functions and a set of constant data members.

Structure

A structure (**struct**) description has TypeKind = TKIND_RECORD. A structure is an empty set of functions and a set of per-instance data members.

Union

A **union** description has TypeKind = TKIND_UNION. A union is an empty set of functions and a set of per-instance data members, each of which has an instance offset of zero.

Module

A **module** has TypeKind = TKIND_MODULE. A module is a set of static functions and a set of static data members.

OLE-Compatible Interface

An **interface** definition has TypeKind = TKIND_INTERFACE. An interface is a set of pure virtual functions and an empty set of data members. If a type description contains any virtual functions, then the pointer to the VTBL is the first 4 bytes of the instance.

The type information fully describes the member functions in the VTBL, including parameter names and types and function return types. It may inherit from no more than one other interface.

With interfaces and dispinterfaces, all members should have different names, except the accessor functions of properties. For property functions having the same name, the documentation string and Help context should be set for only one of the functions (because they define the same property conceptually).

IDispatch Interface

These include objects (TypeKind = TKIND_DISPATCH) that support the **IDispatch** interface with a specification of the dispatch data members (such as properties) and methods supported through the object's **Invoke** implementation. All members of the dispinterface should have different IDs, except for the accessor functions of properties.

Dual Interface

Dual interfaces (**dual**) have two different type descriptions for the same interface. The TKIND_INTERFACE type description describes the interface as a standard OLE Component Object Model (COM) interface. The TKIND_DISPATCH type decription describes the interface as a standard dispatch interface. The *lcid* and *retval* parameters, and the HRESULT return types are removed, and the return type of the member is specified to be the same type as the *retval* parameter.

By default, the TYPEKIND enumeration for a dual interface is TKIND_DISPATCH. Tools that bind to interfaces should check the type flags for TYPEFLAG_FDUAL. If this flag is set, the TKIND_INTERFACE type description is available through a call to **ITypeInfo::GetRefTypeOfImplType** with an *index* of -1, followed by a call to **ITypeInfo::GetRefTypeInfo**.

Component Object Classes

These cloclass objects (TypeKind = TKIND_COCLASS) support a set of implemented interfaces, which can be of either TKIND_INTERFACE or TKIND_DISPATCH.

ITypeInfo::AddressOfMember

HRESULT ITypeInfo::AddressOfMember(

MEMBERID *memid*,
INVOKEKIND *invkind*,
VOID FAR* FAR* *lplpvoid*
);

Retrieves the addresses of static functions or variables, such as those defined in a DLL.

Parameters

memid

Member ID of the static member whose address is to be retrieved. The member ID is defined by the dispatch ID (DISPID).

invkind

Specifies whether the member is a property, and if so, what kind.

lplpvoid

On return, points to a pointer to the static member.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_WRONGTYPEKIND	Type mismatch.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_ELEMENTNOTFOUND	The element was not found.
TYPE_E_DLLFUNCTIONNOTFOUND	The function could not be found in the DLL.
TYPE_E_CANTLOADLIBRARY	The type library or DLL could not be loaded.

Comments

The addresses are valid until the caller releases its reference to the type description. The *invkind* parameter can be ignored unless the address of a property function is being requested.

If the type description inherits from another type description, this function is recursive to the base type

description, if necessary, to find the item with the requested member ID.

ITypeInfo::CreateInstance

HRESULT ITypeInfo::CreateInstance(

IUnknown FAR* *punkOuter*,
REFIID *riid*,
VOID FAR* FAR* *ppvObj*
);

Creates a new instance of a type that describes a component object class (coclass).

Parameters

punkOuter

A pointer to the controlling **IUnknown**. If Null, then a stand-alone instance is created. If valid, then an aggregate object is created.

riid

An ID for the interface that the caller will use to communicate with the resulting object.

ppvObj

On return, points to a pointer to an instance of the created object.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
TYPE_E_WRONGTYPEKIND	Type mismatch.
E_INVALIDARG	One or more of the arguments is invalid.
E_NOINTERFACE	OLE could not find an implementation of one or more required interfaces.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
Other return codes	Additional errors may be returned from **GetActiveObject** or **CoCreateInstance**.

Comments

For types that describe a component object class (coclass), **CreateInstance** creates a new instance of the class. Normally, **CreateInstance** calls **CoCreateInstance** with the type description's globally unique identifier (GUID). For an Application object, it first calls **GetActiveObject**. If the application is active, **GetActiveObject** returns the active object; otherwise, if **GetActiveObject** fails, **CreateInstance** calls **CoCreateInstance**.

ITypeInfo::GetContainingTypeLib

HRESULT ITypeInfo::GetContainingTypeLib(

ITypeLib FAR* FAR* *lplptlib*,
unsigned int FAR* *lpindex*
);

Retrieves the containing type library and the index of the type description within that type library.

Parameters

lplptlib

On return, points to the containing type library.

lpindex

On return, points to the index of the type description within the containing type library.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_NOINTERFACE	OLE could not find an implementation of one or more required interfaces.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.

ITypeInfo::GetDllEntry

HRESULT ITypeInfo::GetDllEntry(

MEMBERID *memid*,
INVOKEKIND *invkind*,
BSTR FAR* *lpbstrDllName*,
BSTR FAR* *lpbstrName*,
unsigned short FAR* *lpwOrdinal*
);

Retrieves a description or specification of an entry point for a function in a DLL.

Parameters

memid

ID of the member function whose DLL entry description is to be returned.

invkind

Specifies the kind of member identified by *memid*. This is important for properties, because one *memid* can identify up to three separate functions.

lpbstrDllName

If not Null, the function sets *lpbstrDllName* to a BSTR that contains the name of the DLL.

lpbstrName

If not Null, the function sets *lpbstrName* to a BSTR that contains the name of the entry point. If the entry point is specified by an ordinal, **lpbstrName* is set to Null.

lpwOrdinal

If not Null, and if the function is defined by an ordinal, then *lpwOrdinal* is set to point to the ordinal.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_NOINTERFACE	OLE could not find an implementation of one or more required interfaces.
TYPE_E_ELEMENTNOTFOUND	The element was not found.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.

TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

The caller passes in a MEMID, which represents the member function whose entry description is desired. If the function has a DLL entry point, the name of the DLL that contains the function, as well as its name or ordinal identifier, are placed in the passed-in pointers allocated by the caller. If there is no DLL entry point for the function, an error is returned.

If the type description inherits from another type description, this function is recursive to the base type description, if necessary, to find the item with the requested member ID.

The caller should use **SysFreeString()** to free the BSTRs referenced by *lpbstrName* and *lpbstrDllName*.

ITypeInfo::GetDocumentation

HRESULT ITypeInfo::GetDocumentation(

MEMBERID *memid*,
BSTR FAR* *lpbstrName*,
BSTR FAR* *lpbstrDocString*,
unsigned long FAR* *lpdwHelpContext*,
BSTR FAR* *lpbstrHelpFile*
);

Retrieves the documentation string, the complete Help file name and path, and the context ID for the Help topic for a specified type description.

Parameters

memid

ID of the member whose documentation is to be returned.

lpbstrName

Pointer to a BSTR allocated by the callee into which the name of the specified item is placed. If the caller does not need the item name, *lpbstrName* can be Null.

lpbstrDocString

Pointer to a BSTR into which the documentation string for the specified item is placed. If the caller does not need the documentation string, *lpbstrDocString* can be Null.

lpdwHelpContext

Pointer to the Help context associated with the specified item. If the caller does not need the Help context, the *lpdwHelpContext* can be Null.

lpbstrHelpFile

Pointer to a BSTR into which the fully qualified name of the Help file is placed. If the caller does not need the Help file name, *lpbstrHelpFile* can be Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_ELEMENTNOTFOUND	The element was not found.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be

	opened.
TYPE_E_ELEMENTNOTFOUND	The element was not found.

Comments

The function **GetDocumentation** provides access to the documentation for the member specified by the *memid* parameter. If the passed-in *memid* is MEMBERID_NIL, then the documentation for the type description is returned.

If the type description inherits from another type description, this function is recursive to the base type description, if necessary, to find the item with the requested member ID.

The caller should use **SysFreeString()** to free the BSTRs referenced by *lpbstrName*, *lpbstrDocString*, and *lpbstrHelpFile*.

Example

```
CHECKRESULT(ptypeinfo->GetDocumentation(idMember, &bstrName, NULL, NULL,
     NULL));
.
.
.
SysFreeString (bstrName);
```

ITypeInfo::GetFuncDesc

HRESULT ITypeInfo::GetFuncDesc(

unsigned int *index*,
FUNCDESC FAR* FAR* *lplpfuncdesc*
);

Retrieves the FUNCDESC structure that contains information about a specified function.

Parameters

index

Index of the function whose description is to be returned. The *index* should be in the range of 0 to 1 less than the number of functions in this type.

lplpfuncdesc

On return, points to a pointer to a FUNCDESC that describes the specified function.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

The function **GetFuncDesc** provides access to a FUNCDESC structure that describes the function with the specified *index*. The FUNCDESC should be freed with **ITypeInfo::ReleaseFuncDesc()**. The number of functions in the type is one of the attributes contained in the TYPEATTR structure.

Example

```
CHECKRESULT(ptypeinfo->GetFuncDesc(i, &pfuncdesc));
idMember = pfuncdesc->elemdescFunc.ID;
CHECKRESULT(ptypeinfo->GetDocumentation(idMember, &bstrName, NULL, NULL,
NULL));
ptypeinfo->ReleaseFuncDesc(pfuncdesc);
```

ITypeInfo::GetIDsOfNames

HRESULT ITypeInfo::GetIDsOfNames(

OLECHAR FAR* FAR* *rgszNames*,
unsigned int *cNames*,
MEMBERID FAR* *rgmemid*
);

Maps between member names and member IDs, and parameter names and parameter IDs.

Parameters

rgszNames

Passed-in pointer to an array of names to be mapped.

cNames

Count of the names to be mapped.

rgmemid

Caller-allocated array in which name mappings are placed.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
DISP_E_UNKNOWNNAME	One or more of the names could not be found.
DISP_E_UNKNOWNLCID	The locale ID could not be found in the OLE DLLs.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

The function **GetIDsOfNames** maps the name of a member (*rgszNames*[0]) and its parameters (*rgszNames*[1] ...*rgszNames*[*cNames* - 1]) to the ID of the member (*rgid*[0]), and to the IDs of the specified parameters (*rgid*[1] ... *rgid*[*cNames* - 1]). The IDs of parameters are 0 for the first parameter in the member function's argument list, 1 for the second, and so on.

If the type description inherits from another type description, this function is recursive to the base type description, if necessary, to find the item with the requested member ID.

ITypeInfo::GetImplTypeFlags

HRESULT ITypeInfo:: GetImplTypeFlags(

unsigned int *index*,
int* *pimpltypeflags*
);

Retrieves the IMPLTYPEFLAGS enumeration for one implemented interface or base interface in a type description.

Parameters

index

Index of the implemented interface or base interface for which to get the flags.

pimpltypeflags

On return, pointer to the IMPLTYPEFLAGS enumeration.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

The flags are associated with the act of inheritance, and not with the inherited interface.

ITypeInfo::GetMops

HRESULT ITypeInfo::GetMops(

MEMBERID *memid*,
BSTR FAR* *lpbstrMops*
);

Retrieves marshaling information.

Parameters

memid

The member ID that indicates which marshaling information is needed.

lpbstrMops

On return, contains a pointer to the *opcode* string used in marshaling the fields of the structure described by the referenced type description, or returns Null if there is no information to return.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_ELEMENTNOTFOUND	The element was not found.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

If the passed-in member ID is MEMBERID_NIL, the function returns the *opcode* string for marshaling the fields of the structure described by the type description. Otherwise, it returns the *opcode* string for marshaling the function specified by the *index*.

If the type description inherits from another type description, this function recurses on the base type description, if necessary, to find the item with the requested
member ID.

ITypeInfo::GetNames

HRESULT ITypeInfo::GetNames(

MEMBERID *memid*,
BSTR FAR* *rgbstrNames*,
unsigned int *cNameMax*,
unsigned int FAR* *lpcName*
);

Retrieves the variable with the specified member ID (or the name of the property or method and its parameters) that correspond to the specified function ID.

Parameters

memid

The ID of the member whose name (or names) is to be returned.

rgbstrNames

Pointer to the caller-allocated array. On return, each of these *lpcName* elements is filled in to point to a BSTR that contains the name (or names) associated with the member.

cNameMax

Length of the passed-in *rgbstrNames* array.

lpcName

On return, points to the number that represents the number of names in *rgbstrNames* array.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.
TYPE_E_ELEMENTNOTFOUND	The element was not found.

Comments

The caller must release the returned BSTR (Basic string) array.

If the member ID identifies a property that is implemented with property functions, the property name is

returned.

For property **get** functions, the names of the function and its parameters are always returned.

For property put and put reference functions, the right side of the assignment is unnamed. If *cNameMax* is less than is required to return all of the names of the parameters of a function, then only the names of the first *cNameMax* - 1 parameters are returned. The names of the parameters are returned in the array in the same order that they appear elsewhere in the interface (for example, the same order in the parameter array associated with the FUNCDESC enumeration).

If the type description inherits from another type description, this function is recursive to the base type description, if necessary, to find the item with the requested member ID.

ITypeInfo::GetRefTypeInfo

HRESULT ITypeInfo::GetRefTypeInfo(*hreftype*, *lplptinfo***)**
HREFTYPE *hreftype*
ITypeInfo FAR* FAR* *lplptinfo*

If a type description references other type descriptions, it retrieves the referenced type descriptions.

Parameters

hreftype

Handle to the referenced type description to be returned.

lplptinfo

Points a pointer to a pointer to the referenced type description.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.
TYPE_E_ELEMENTNOTFOUND	The element was not found.
TYPE_E_REGISTRYACCESS	There was an error accessing the system registration database.
TYPE_E_LIBNOTREGISTERED	The type library was not found in the system registration database.

Comments

On return, the second parameter contains a pointer to a pointer to a type description that is referenced by this type description. A type description must have a reference to each type description that occurs as the type of any of its variables, function parameters, or function return types. For example, if the type of a data member is a record type, the type description for that data member contains the *hreftype* of a referenced type description. To get a pointer to the type description, the reference is passed to **GetRefTypeInfo**.

ITypeInfo::GetRefTypeOfImplType

HRESULT ITypeInfo::GetRefTypeOfImplType(

unsigned int *index*,
HREFTYPE FAR* *lphreftype*
);

If a type description describes a Component Object Model (COM) class, it retrieves the type description of the implemented interface types. For an interface, **GetRefTypeOfImplType** returns the type information for inherited interfaces, if any exist.

Parameters

index

Index of the implemented type whose handle is returned. The valid range is 0 to the *cImplTypes* field in the TYPEATTR structure.

lphreftype

On return, points to a handle for the implemented interface (if any). This handle can be passed to **ITypeInfo::GetRefTypeInfo** to get the type description.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
TYPE_E_ELEMENTNOTFOUND	Passed index is outside the range 0 to 1 less than the number of function descriptions.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

If the TKIND_DISPATCH type description is for a dual interface, the TKIND_INTERFACE type description can be obtained by calling **GetRefTypeOfImplType** with an *index* of -1, and by passing the returned *lphreftype* handle to **GetRefTypeInfo** to retrieve the type information.

ITypeInfo::GetTypeAttr

HRESULT ITypeInfo::GetTypeAttr(

TYPEATTR FAR* FAR* *lplptypeattr*
);

Retrieves a TYPEATTR structure that contains the attributes of the type description.

Parameter

lplptypeattr

On return, points to a pointer to a structure that contains the attributes of this type description.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

To free the TYPEATTR structure, use **ITypeInfo::ReleaseTypeAttr**.

Example

```
CHECKRESULT(ptypeinfoCur->GetTypeAttr(&ptypeattrCur));
.
.
.
ptypeinfoCur->ReleaseTypeAttr(ptypeattrCur);
```

ITypeInfo::GetTypeComp

HRESULT ITypeInfo::GetTypeComp(

ITypeComp FAR* FAR* *lplpcomp*
);

Retrieves the **ITypeComp** interface for the type description, which enables a client compiler to bind to the type description's members.

Parameter

lplpcomp

On return, points to a pointer to the **ITypeComp** of the containing type library.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

A client compiler can use the **ITypeComp** interface to bind to members of the type.

ITypeInfo::GetVarDesc

HRESULT ITypeInfo::GetVarDesc(

unsigned int *index*,
VARDESC FAR* FAR* *lplpvardesc*
);

Retrieves a VARDESC structure that describes the specified variable.

Parameters

index

Index of the variable whose description is to be returned. The *index* should be in the range of 0 to 1 less than the number of variables in this type.

lplpvardesc

On return, points to a pointer to a VARDESC that describes the specified variable.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

To free the VARDESC structure, use **ReleaseVarDesc.**

Example

```
CHECKRESULT(ptypeinfo->GetVarDesc(i, &pvardesc));
idMember = pvardesc->memid;
CHECKRESULT(ptypeinfo->GetDocumentation(idMember, &bstrName, NULL, NULL,
           NULL));
ptypeinfo->ReleaseVarDesc(pvardesc);
```

ITypeInfo::Invoke

HRESULT ITypeInfo::Invoke(

VOID FAR* *lpvInstance*,
MEMBERID *memid*,
unsigned short *wFlags*,
DISPPARAMS FAR* *pdispparams*,
VARIANT FAR* *pvargResult*,
EXCEPINFO FAR* *pexcepinfo*,
unsigned int FAR* *puArgErr*
);

Invokes a method, or accesses a property of an object, that implements the interface described by the type description.

Parameters

lpvInstance

Pointer to an instance of the interface described by this type description.

memid

Identifies the interface member.

wFlags

Flags describing the context of the invoke call, as follows:

Value	Description
DISPATCH_METHOD	The member is accessed as a method. If there is ambiguity, both this and the DISPATCH_PROPERTYGET flag can be set.
DISPATCH_PROPERTYGET	The member is retrieved as a property or data member.
DISPATCH_PROPERTYPUT	The member is changed as a property or data member.
DISPATCH_PROPERTYPUTREF	The member is changed by using a reference assignment, rather than a value assignment. This value is only valid when the property accepts a reference to an object.

pdispparams

Points to a structure that contains an array of arguments, an array of dispatch IDs (DISPIDs) for named arguments, and counts of the number of elements in each array.

pvargResult

Should be Null if the caller does not expect any result. Otherwise, it should be a pointer to the location at which the result is to be stored. If *wFlags* specifies DISPATCH_PROPERTYPUT or DISPATCH_PROPERTYPUTREF, *pvargResult* is ignored.

pexcepinfo

Points to an exception information structure, which is filled in only if DISP_E_EXCEPTION is returned. If *pexcepinfo* is Null on input, only an HRESULT error will be returned.

puArgErr

If **Invoke** returns DISP_E_TYPEMISMATCH, *puArgErr* indicates the index (within *rgvarg*) of the argument with incorrect type. If more than one argument returns an error, *puArgErr* indicates only the first argument with an error. Arguments in *pdispparams->rgvarg* appear in reverse order, so the first argument is the one having the highest index in the array. Cannot be Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_INVALIDARG	One or more of the arguments is invalid.
DISP_E_EXCEPTION	The member being invoked has returned an error HRESULT. If the member implements **IErrorInfo**, details are available in the error object. Otherwise, the *pexcepinfo* parameter contains details.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_REGISTRYACCESS	There was an error accessing the system registration database.
TYPE_E_LIBNOTREGISTERED	The type library was not found in the system registration database.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_WRONGTYPEKIND	Type mismatch.
TYPE_E_ELEMENTNOTFOUND	The element was not found.
TYPE_E_BADMODULEKIND	The module does not support **Invoke**.
Other return codes	Any of the **IDispatch::Invoke** errors may also be returned.

Comments

Use the function **ITypeInfo::Invoke** to access a member of an object or invoke a method that implements the interface described by this type description. For objects that support the **IDispatch** interface, you can use **Invoke** to implement **IDispatch::Invoke**.

ITypeInfo::Invoke takes a pointer to an instance of the class. Otherwise, its parameters are the same as **IDispatch::Invoke,** except that **ITypeInfo::Invoke** omits the *refiid* and *lcid* parameters. When called, **ITypeInfo::Invoke** performs the actions described by the **IDispatch::Invoke** parameters on the specified instance.

For VTBL interface members, **ITypeInfo::Invoke** passes the locale ID (LCID) of the type information into parameters tagged with the **lcid** attribute, and the returned value into the **retval** attribute.

If the type description inherits from another type description, this function recurses on the base type description to find the item with the requested member ID.

ITypeInfo::ReleaseFuncDesc

VOID ITypeInfo::ReleaseFuncDesc(

FUNCDESC FAR* *lpfuncdesc*
);

Releases a FUNCDESC previously returned by **GetFuncDesc**.

Parameter

lpfuncdesc

Pointer to the FUNCDESC to be freed.

Comments

The function **ReleaseFuncDesc** releases a FUNCDESC that was returned through **ITypeInfo::GetFuncDesc**.

Example

```
ptypeinfoCur->ReleaseFuncDesc(pfuncdesc);
```

ITypeInfo::ReleaseTypeAttr

VOID ITypeInfo::ReleaseTypeAttr(

TYPEATTR FAR* *lptypeattr*
);

Releases a TYPEATTR previously returned by **GetTypeAttr**.

Parameter

lptypeattr

Pointer to the TYPEATTR to be freed.

Comments

The function **ReleaseTypeAttr** releases a TYPEATTR that was returned through **ITypeInfo::GetTypeAttr**.

ITypeInfo::ReleaseVarDesc

VOID ITypeInfo::ReleaseVarDesc(

VARDESC FAR* *lpvardesc*
);

Releases a VARDESC previously returned by **GetVarDesc**.

Parameter

lpvardesc

Pointer to the VARDESC to be freed.

Comments

ReleaseVarDesc releases a VARDESC that was returned through **ITypeInfo::GetVarDesc**.

Example

```
VARDESC          FAR *pvardesc;
CHECKRESULT(ptypeinfo->GetVarDesc(i, &pvardesc));
idMember = pvardesc->memid;
CHECKRESULT(ptypeinfo->GetDocumentation(idMember, &bstrName, NULL, NULL,
     NULL));
ptypeinfo->ReleaseVarDesc(pvardesc);
```

New Automation Interfaces

There are two new interfaces for Automation:

ITypeInfo2::ITypeInfo

ITypeLib2::ITypeLib

Because they inherit from **ITypeInfo** and **ITypeLib,** an **ITypeInfo** can be cast to an **ITypeInfo2** instead of using the calls **QueryInterface()** and **Release()**.

By adding the the new methods described in the following section, **QueryInterface** can be called to **ITypeInfo2** and **ITypeLib2** in the same way as **ITypeInfo** and **ITypeLib**.

Data Access

The following methods can be used to retrieve data efficiently.

ITypeInfo2::GetTypeKind

```
HRESULT ITypeInfo2::GetTypeKind(TYPEKIND * ptypekind)
```

The **GetTypeKind()** method returns the TYPEKIND enumeration without any allocations.

ITypeInfo2::GetTypeFlags

```
HRESULT ITypeInfo2::GetTypeFlags(DWORD * pdwTypeFlags)
```

Returns the type flags without any allocations. This returns a DWORD type flag, which expands the type flags without growing the TYPEATTR (type attribute).

ITypeInfo2::GetFuncIndexOfMemId

```
ITypeInfo2::GetFuncIndexOfMemId(
MEMID memid,
INVOKEKIND invkind,
UINT * pfuncIndex)
```

This method allows binding to **VBA 5.0** to a specific member, based on a known DISPID when the member name is not known (for example, when binding to the default member).

ITypeInfo2::GetVarIndexOfMemId

```
ITypeInfo2::GetVarIndexOfMemId(MEMID memid, UINT * pvarIndex)
```

Returns the index into the spcified memory identifier.

ITypeLib2::GetLibStatistics

```
ITypeLib2::GetLibStatistics(DWORD *pcUniqueNames, DWORD * pcchUniqueNames)
```

Returns statitistics about the type library that are required for efficient sizing of hash tables in VBA's compiler.

ITypeLib2 Interface

The **ITypeLib2** interface inherits from the **ITypeLib** interface. This allows **ITypeLib** to cast to an **ITypeLib2** in performance-sensitive cases, rather than performing extra **QueryInterface()** and **Release()** calls.

Example

```
DECLARE_INTERFACE_(ITypeLib2, ITypeLib)
{
BEGIN_INTERFACE
```

ITypeLib2::GetCustData

HRESULT GetCustData(

  REFGUID *guid*,
  VARIANT **pVarVal*
);

Gets the custom data.

Parameter

guid

Globally unique ID (GUID) used to identify the data.

pVarVal

Where to put the retrieved data.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeLib2::GetDocumentation2

HRESULT ITypeLib2::GetDocumentation2(

[in] int *index,*
[in] LCID *lcid*,
[out] BSTR FAR* *lpbstrHelpString*,
[out] unsigned long FAR* *lpdwHelpStringContext*,
BSTR FAR* *lpbstrHelpStringDll*
);

Retrieves the library's documentation string, the complete Help file name and path, the localization context to use, and the context ID for the library Help topic in the Help file.

Parameters

index

Index of the type description whose documentation is to be returned; if *index* is -1, then the documentation for the library is returned.

lcid

Locale identifier.

lpbstrHelpString

Returns a BSTR that contains the name of the specified item. If the caller does not need the item name, then *lpbstrName* can be Null.

lpdwHelpStringContext

Returns the Help localization context. If the caller does not need the Help context, then it can be Null.

lpbstrHelpStringDll

Returns a BSTR that contains the fully qualified name of the file containing the DLL used for Help file. If the caller does not need the file name, then it can be Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_ELEMENTNOTFOU	The element was not found.

ND

Comments

Gets information at the type library level. The caller should free the BSTR parameters.

This function will call **_DLLGetDocumentation** in the specified DLL to retrieve the desired Help string, if there is a Help string context for this item. If no Help string context exists or an error occurs, then it will defer to the **GetDocumentation** method and return the associated documentation string.

ITypeLib2::GetLibStatistics

HRESULT GetLibStatistics(

  unsigned long* *pcUniqueNames*,
  unsigned long* *pcchUniqueNames*
);

Returns statistics about a type library that are required for efficient sizing of hash tables.

Parameter

pcUniqueNames

Returns a pointer to a count of unique names.

pcchUniqueNames

Returns a pointer to a change in the count of unique names.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeLib2::GetHelpStringContext

HRESULT GetHelpStringContext(

unsigned int *index*,
unsigned long **pdwHelpStringContext*
);

Gets the context number for the specified Help string.

Parameter

index

Index of the Help string.

pdwHelpStringContext

Help string context number (DWORD).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetTypeKind

HRESULT GetTypeKind(

TYPEKIND **pTypeKind*
);

Returns the TYPEKIND enumeration quickly, without doing any allocations.

Parameter

pTypeKind

Reference to a TYPEKIND enumeration.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is

invalid.

ITypeInfo2::GetTypeFlags

HRESULT GetTypeFlags(

unsigned long **pdwTypeFlags*
);

Returns the TYPEFLAGS quickly, without doing any allocations. This method returns a DWORD TYPEFLAG.

Parameter

pdwTypeFlags

The DWORD reference to a TYPEFLAG.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetFuncIndexOfMemId

HRESULT GetFuncIndexOfMemId(

MEMBERID *memid*,
INVOKEKIND *invkind*,
unsigned int **pFuncIndex*
);

Binds to a specific member based on a known dispatch ID (DISPID), where the member name is not known (for example, when binding to a default member).

Parameter

memid

Member identifier.

invkind

Invoke kind.

pFuncIndex

Returns an index into the function.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetVarIndexOfMemId

```
HRESULT GetVarIndexOfMemId(
  MEMBERID   memid,
  unsigned int   *pVarIndex
);
```

Binds to a specific member based on a known DISPID, where the member name is not known (for example, when binding to a default member).

Parameter

memid

Member identifier.

pVarIndex

Returns the index.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetFuncCustData

HRESULT GetFuncCustData(

unsigned int *index*,
REFGUID *guid*,
VARIANT **pVarVal*
);

Gets the custom data from the specified function.

Parameter

index

The index of the function for which to get the custom data.

guid

The globally unique ID (GUID) used to identify the data.

pVarVal

Where to put the data.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetParamCustData

```
HRESULT GetParamCustData(
  unsigned int   indexFunc,
  unsigned int   indexParam,
  REFGUID   guid,
  VARIANT   *pVarVal
);
```

Gets the specified custom data parameter.

Parameter

indexFunc

Index of the function for which to get the custom data.

IndexParam

Index of the parameter of this function for which to get the custom data.

guid

Globally unique ID (GUID) used to identify the data.

pVarVal

Where to put the retrieved data.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetVarCustData

HRESULT GetVarCustData(

unsigned int *index*,
REFGUID *guid*,
VARIANT **pVarVal*
);

Gets the variable for the custom data.

Parameter

index

Index of the variable for which to get the custom data.

guid

Globally unique ID (GUID) used to identify the data.

PVarVal

Where to put the retrieved data.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetImplTypeCustData

```
HRESULT GetImplTypeCustData(
  unsigned int   index,
  REFGUID   guid,
  VARIANT   *pVarVal
);
```

Gets the implementation type of the custom data.

Parameters

index

Index of the implementation type for the custom data.

guid

GUID used to identify the data.

pVarVal

Where to put the retrieved data.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeInfo2::GetDocumentation2

HRESULT ITypeInfo2::GetDocumentation2(

[in] MEMID *memid,*
[in] LCID *lcid*,
[out] BSTR FAR* *lpbstrHelpString*,
[out] unsigned long FAR* *lpdwHelpStringContext*,
BSTR FAR* *lpbstrHelpStringDll*
);

Retrieves the documentation string, the complete Help file name and path, the localization context to use, and the context ID for the library Help topic in the Help file.

Parameters

memid

Member identifier for the type description.

lcid

Locale identifier (LCID).

lpbstrHelpString

Returns a BSTR that contains the name of the specified item. If the caller does not need the item name, then *lpbstrName* can be Null.

lpdwHelpStringContext

Returns the Help localization context. If the caller does not need the Help context, it can be Null.

lpbstrHelpStringDll

Returns a BSTR that contains the fully qualified name of the file containing the DLL used for Help file. If the caller does not need the file name, it can be Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_ELEMENTNOTFOUND	The element was not found.

Comments

Gets information at the type information level (about the type information and its members). The caller should free the BSTR parameters.

This function will call **_DLLGetDocumentation** in the specified DLL to retrieve the desired Help string, if there is a Help string context for this item. If no Help string context exists or an error occurs, then it will defer to the **GetDocumentation** method and return the associated documentation string.

ITypeInfo2::GetHelpStringContext

HRESULT GetHelpStringContext(

unsigned int *index*,
unsigned long **pdwHelpStringContext*
);

Gets the context number for the specified Help string.

Parameter

index

Index of the Help string.

pdwHelpStringContext

Help string context number (DWORD).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeComp Interface

The **ITypeComp** interface provides a fast way to access information that compilers need when binding to and instantiating structures and interfaces. Binding is the process of mapping names to types and type members.

Implemented by	Used by	Header file name
Oleaut32.dll (32-bit systems) Typelib.dll (16-bit systems)	Tools that need to access the descriptions of objects contained in type libraries.	Oleauto.h Dispatch.h

ITypeComp::Bind

HRESULT ITypeComp::Bind(

OLECHAR FAR* *szName*,
unsigned long *lHashVal*,
unsigned short *wFlags*,
ITypeInfo FAR*FAR* *lplptinfo*,
DESCKIND FAR* *lpdesckind*,
BINDPTR FAR* *lpbindptr*
);

Maps a name to a member of a type, or binds global variables and functions contained in a type library.

Parameters

szName

Name to be bound.

lHashVal

Hash value for the name computed by **LHashValOfNameSys**.

wFlags

Flags word containing one or more of the **Invoke** flags defined in the INVOKEKIND enumeration. Specifies whether the name was referenced as a method or a property. When binding to a variable, specify the flag INVOKE_PROPERTYGET. Specify zero to bind to any type of member.

lplptinfo

If a FUNCDESC or VARDESC was returned, then *lplptinfo* points to a pointer to the type description that contains the item to which it is bound.

lpdesckind

Pointer to a DESCKIND enumerator that indicates whether the name bound to is a VARDESC, FUNCDESC, or TYPECOMP. If there was no match, points to DESCKIND_NONE.

lpbindptr

On return, contains a pointer to the bound-to VARDESC, FUNCDESC, or **ITypeComp** interface.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.

TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_AMBIGUOUSNAME	More than one instance of this name occurs in the type library.

Comments

Use **Bind** for binding to the variables and methods of a type, or for binding to the global variables and methods in a type library. The returned DESCKIND pointer *lpdesckind* indicates whether the name was bound to a VARDESC, a FUNCDESC, or to an **ITypeComp** instance. The returned *lpbindptr* points to the VARDESC, FUNCDESC, or **ITypeComp**.

If a data member or method is bound to, then *lplptinfo* points to the type description that contains the method or data member.

If **Bind** binds the name to a nested binding context, it returns a pointer to an **ITypeComp** instance in *lpbindptr* and a Null type description pointer in *lplptinfo*. For example, if the name of a type description is passed for a module (TKIND_MODULE), enumeration (TKIND_ENUM), or coclass (TKIND_COCLASS), **Bind** returns the **ITypeComp** instance of the type description for the module, enumeration, or coclass. This feature supports languages such as Visual Basic that allow references to members of a type description to be qualified by the name of the type description. For example, a function in a module can be referenced by *modulename.functionname*.

The members of TKIND_ENUM, TKIND_MODULE, and TKIND_COCLASS types marked as Application objects can be bound to directly from **ITypeComp,** without specifying the name of the module. The **ITypeComp** of a coclass defers to the **ITypeComp** of its default interface.

As with other methods of **ITypeComp**, **ITypeInfo**, and **ITypeLib**, the calling code is responsible for releasing the returned object instances or structures. If a VARDESC or FUNCDESC is returned, the caller is responsible for deleting it with the returned type description and releasing the type description instance itself. Otherwise, if an **ITypeComp** instance is returned, the caller must release it.

Special rules apply if you call a type library's **Bind** method, passing it the name of a member of an Application object class (a class that has the TYPEFLAG_FAPPOBJECT flag set). In this case, **Bind** returns DESCKIND_IMPLICITAPPOBJ in *lpdesckind,* a VARDESC that describes the Application object in *lpbindptr*, and the **ITypeInfo** of the Application object class in *lplptinfo*. To bind to the object, **ITypeInfo::GetTypeComp** must make a call to get the **ITypeComp** of the Application object class, and then reinvoke its **Bind** method with the name initially passed to the type library's **ITypeComp**.

The caller should use the returned **ITypeInfo** pointer (*lplptinfo*) to get the address of the member.

Note The *wflags* parameter is the same as the *wflags* parameter in **IDispatch::Invoke**.

ITypeComp::BindType

HRESULT ITypeComp::BindType(

OLECHAR FAR* *szName*,
unsigned long *lHashVal*,
ITypeInfo FAR* FAR* *lplptinfo*,
ITypeComp FAR* FAR* *lplptcomp*
);

Binds to the type descriptions contained within a type library.

Parameters

szName

Name to be bound.

lHashVal

Hash value for the name computed by **LHashValOfName**.

lplptinfo

On return, contains a pointer to a pointer to an **ITypeInfo** of the type to which the name was bound.

lplptcomp

Passes a valid pointer, such as the address of an **ITypeComp*** variable.

Example

```
TypeComp * ptcomp;
      ptemp -> BindType(szName, lhashval, &ptinfo, &ptemp)
```

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVDATAREAD	Invalid data.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_AMBIGUOUSNAME	More than one instance of this name occurs in the type library.

Comments

Use the function **BindType** for binding a type name to the **ITypeInfo** that describes the type. This function

is invoked on the **ITypeComp** that is returned by **ITypeLib::GetTypeComp** to bind to types defined within that library. It can also be used in the future for binding to nested types.

Overview of Type Compilation and Library Functions

The functions for loading, registering, and querying type libraries are provided by Oleaut32.dll (for 32-bit systems) and Typelib.dll (for 16-bit systems).

Category	Function name	Purpose
Library loading	**LoadTypeLib**	Loads and registers a type library.
	LoadRegTypeLib	Uses registry information to load a type library.
Library registration	**RegisterTypeLib**	Adds information about a type library to the system registry.
	UnRegisterTypeLib	Removes type library information added through **RegisterTypeLib** to allow uninstall procedures.
	LoadTypeLibEx	Loads a type library and (optionally) registers it in the system registry
	QueryPathOfRegTypeLib	Retrieves the path of a registered type library.
Type compilation	**LHashValOfNameSys** **LHashValOfName**	Computes a hash value for a name that can then be passed to **ITypeComp::Bind**, **ITypeComp::BindType**, **ITypeLib::IsName**, or **ITypeLib::FindName**.

LHashValOfName

unsigned long LHashValOfName(

LCID *lcid*,
OLECHAR FAR* *szName*
);

Computes a hash value for a name that can then be passed to **ITypeComp::Bind**, **ITypeComp::BindType**, **ITypeLib::FindName**, or **ITypeLib::IsName**.

Parameters

lcid

The locale ID for the string.

szName

String whose hash value is to be computed.

Return Value

A 32-bit hash value that represents the passed-in name.

Comments

This function is equivalent to **LHashValOfNameSys**. The header file Oleauto.h contains macros that define **LHashValOfName** as **LHashValOfNameSys,** with the target operating system (*syskind*) based on the build preprocessor flags.

LHashValOfName computes a 32-bit hash value for a name that can be passed to **ITypeComp::Bind**, **ITypeComp::BindType**, **ITypeLib::FindName**, or **ITypeLib::IsName**. The returned hash value is independent of the case of the characters in *szName*, as long as the language of the name is one of the languages supported by the OLE National Language Specification API. Any two strings that match when a case-insensitive comparison is done using any language produce the same hash value.

LHashValOfNameSys Quick Info

unsigned long LHashValOfNameSys(

SYSKIND *syskind*,
LCID *lcid*,
OLECHAR FAR* *szName*
);

Computes a hash value for a name that can then be passed to **ITypeComp::Bind**, **ITypeComp::BindType**, **ITypeLib::FindName**, or **ITypeLib::IsName**.

Parameters

syskind

The SYSKIND of the target operating system.

lcid

The locale ID for the string.

szName

String whose hash value is to be computed.

Return Value

A 32-bit hash value that represents the passed-in name.

LoadTypeLib Quick Info

HRESULT LoadTypeLib(

OLECHAR FAR* *szFileName*,
ITypeLib FAR* FAR* *lplptlib*
);

Loads and registers a type library.

Parameters

szFileName

Contains the name of the file from which **LoadTypeLib** should attempt to load a type library.

lplptlib

On return, contains a pointer to a pointer to the loaded type library.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_UNKNOWNLCID	The locale ID could not be found in the OLE-supported DLLs.
TYPE_E_CANTLOADLIBRARY	The type library or DLL could not be loaded.
Other return codes	All FACILITY_STORAGE errors can be returned.

Comments

The function **LoadTypeLib** loads a type library (usually created with MkTypeLib) that is stored in the specified file. If *szFileName* specifies only a file name without any path, **LoadTypeLib** searches for the file and proceeds as follows:

- If the file is a stand-alone type library implemented by Typelib.dll, the library is loaded directly.
- If the file is a DLL or an executable file, it is loaded. By default, the type library is extracted from the first resource of type **ITypeLib**. To load a different type of library resource, append an integer index to *szFileName*. For example:

```
LoadTypeLib("C:\MONTANA\EXE\MFA.EXE\3", lplptlib)
```

This statement loads the type library resource 3 from the file Mfa.exe file.

- If the file is none of the above, the file name is parsed into a moniker (an object that represents a file-based link source), and then bound to the moniker. This approach allows **LoadTypeLib** to be used on foreign type libraries, including in-memory type libraries. Foreign type libraries cannnot reside in a DLL or an executable file. For more information on monikers, see the *OLE Programmer's Guide and Reference* in the Win32 SDK.

If the type library is already loaded, **LoadTypeLib** increments the type library's reference count and returns a pointer to the type library.

For backward compatibility, **LoadTypeLib** will register the type library if the path is not specified in the *szFileName* parameter. **LoadTypeLib** will not register the type library if the path of the type library is specified. It is recommended that **RegisterTypeLib** be used to register a type library.

LoadRegTypeLib Quick Info

HRESULT LoadRegTypeLib(

REFGUID *guid*,
unsigned short *wVerMajor*,
unsigned short *wVerMinor*,
LCID *lcid*,
ITypeLib FAR* FAR* *lplptlib*
);

Uses registry information to load a type library.

Parameters

guid

The globally unique ID of the library being loaded.

wVerMajor

Major version number of the library being loaded.

wVerMinor

Minor version number of the library being loaded.

lcid

National language code of the library being loaded.

lplptlib

On return, points to a pointer to the loaded type library.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not read from the file.
TYPE_E_INVALIDSTATE	The type library could not be opened.
TYPE_E_INVDATAREAD	The function could not read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_UNKNOWNLCID	The passed in local ID (LCID) could not be found in the OLE-supported DLLs.
TYPE_E_CANTLOADLIBRARY	The type library or DLL could not be loaded.
Other return codes	All FACILITY_STORAGE and system

registry errors can also be returned.

Comments

The function **LoadRegTypeLib** defers to **LoadTypeLib** to load the file.

LoadRegTypeLib compares the requested version numbers against those found in the system registry, and takes one of the following actions:

- If one of the registered libraries exactly matches both the requested major and minor version numbers, then that type library is loaded.
- If one or more registered type libraries exactly match the requested major version number, and has a greater minor version number than that requested, the one with the greatest minor version number is loaded.
- If none of the registered type libraries exactly match the requested major version number (or if none of those that do exactly match the major version number also have a minor version number greater than or equal to the requested minor version number), then **LoadRegTypeLib** returns an error.

RegisterTypeLib Quick Info

HRESULT RegisterTypeLib(

ITypeLib FAR* *ptlib*,
OLECHAR FAR* *szFullPath*,
OLECHAR FAR* *szHelpDir*
);

Adds information about a type library to the system registry.

Parameters

ptlib

Pointer to the type library being registered.

szFullPath

Fully qualified path specification for the type library being registered.

szHelpDir

Directory in which the Help file for the library being registered can be found. Can be Null.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_REGISTRYACCESS	The system registration database could not be opened.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

The function **RegisterTypeLib** can be used during application initialization to register the application's type library correctly.

In addition to filling in a complete registry entry under the type library key, **RegisterTypeLib** adds entries for each of the dispinterfaces and Automation-compatible interfaces, including dual interfaces. This information is required to create instances of these interfaces.

UnRegisterTypeLib

HRESULT UnRegisterTypeLib(

REFGUID *guid*,
unsigned short *wVerMajor*,
unsigned short *wVerMinor*,
LCID *lcid*,
SYSKIND *syskind*
);

Removes type library information from the system registry. Use this API to allow applicaitons to properly uninstall themselves. In-process objects typically call this API from **DllUnregisterServer**.

Parameters

guid

Globally unique identifier.

wVerMajor

Major version number of the type library being removed.

wVerMinor

Minor version number of the type library being removed.

lcid

Locale identifier.

syskind

The target operating system (SYSKIND).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_REGISTRYACCESS	The system registration database could not be opened.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

In-process objects typically call this API from **DllUnregisterServer**.

LoadTypeLibEx

HRESULT LoadTypeLibEx(

LPCOLESTR *szFile,*
REGKIND *regkind*,
ITYPELIB *pptlib*
);

Loads a type library and (optionally) registers it in the system registry.

Parameters

szFile

Specification for the type library file.

regkind

Identifies the kind of registration to perform for the type library (DEFAULT, REGISTER, or NONE).

```
typedef enum tagREGKIND
{
    REGKIND_DEFAULT,
    REGKIND_REGISTER,
    REGKIND_NONE
} REGKIND;
```

pptlib

Reference to the type library being loaded.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not write to the file.
TYPE_E_REGISTRYACCESS	The system registration database could not be opened.
TYPE_E_INVALIDSTATE	The type library could not be opened.

Comments

Enables programmers to specify whether or not the type library should be loaded.

QueryPathOfRegTypeLib Quick Info

HRESULT QueryPathOfRegTypeLib(

REFGUID *guid*,
unsigned short *wVerMajor*,
unsigned short *wVerMinor*,
LCID *lcid*,
LPBSTR *lpBstrPathName*
);

Retrieves the path of a registered type library.

Parameters

guid

Globally unique ID of the library whose path is to be queried.

wVerMajor

Major version number of the library whose path is to be queried.

wVerMinor

Minor version number of the library whose path is to be queried.

lcid

National language code for the library whose path is to be queried.

lpBstrPathName

Caller-allocated BSTR in which the type library name is returned.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.

Comments

Returns the fully qualified file name that is specified for the type library in the registry. The caller allocates the BSTR that is passed in, and must free it after use.

Debugging the Type Library and Type Information

It is now much easier to debug type library and type information leaks in the new format of type libraries. The following information applies to both Win32 and to the Apple Power Macintosh.

- To identify the name of the object, expand the CTypeLib2 or CTypeInfo2 portion of the object variable.
- For type libraries, look for a member variable *m_bstrFileName*. This is a BSTR (unicode on Win32) that contains the path name of the loaded type library.
- For type information, look for a member variable *m_szDebName*. This is the type information name (in ANSI) that is only present when using the debug version of Automation.

Memory Leak Debugging

If a type information and/or a type library is leaked, and the debug version of Automation is being used, Automation will display an assert dialog at process termination for each leak that has been detected.

In the older format, the leaks of type libraries only returned the type information and type library name of the leak. Leaks were also tracked down by using the standard **IMalloc** leak-detection technique.

In the new format, type information for a type library now comes out of a single **IMalloc** allocation, so this technique will not work. The leak asserts for the new format now provide more information. An assert from the file Leak.cpp defines the following:

```
Automation has determined that the application has leaked the following:
Typeinfo: <typeinfo name> (iInfo = X) of typelib <typelibname>,
cRefs = Y, g_OAALLOC = Z
```

The information from this assert is the type information (TypeInfo) name, type library (TypeLib) name, iInfo (index of this TypeInfo in the TypeLib), cRefs (refcount of the object that leaked), and *g_OAAALLOC* (the value of the variable *g_OAALLOC* at the time this object was first instantiated).

Proceed as follows:

1. Make note of the value of the *g_OAALLOC* variable.
2. Set a passcount breakpoint at **DebOAALLOC()**, with this value minus 1.
3. Re-run the scenario that leaked under the debugger. When stopped, the routine creating an instance of this object for the first time will be displayed (such as **ITypeLib::GetTypeInfo()** or **ITypeInfo::GetRefTypeInfo()**).
4. Trace out of Automation before checking the name (it is not always set before the passcount is updated), and then watch what happens to this object in order to track down the leak. It's a good idea to double-check that the name of this object is the same as the one originally reported in the leak to make sure there wasn't an error in the passcount.

Breakpoints can be added to **CTypeInfo2::AddRef()** and **Release()**, and calls can be watched for a particular type information. Type library leaks are handled in the same way. The text of the message is slightly different, but it also includes the *g_OAALLOC* count.

Type Library Performance

The performance of type libraries is critical to the overall performance of many components and applications produced at Microsoft. Type libraries are central to the Automation technology and to VBA 5.0.

- At Form load time in VBA 5.0, several type libraries and type information are potentially loaded, depending on the functionality of the form being requested.
- At startup time (host or Visual Basic), the host's type library is loaded as part of the initialization sequence.
- Whenever an **ActiveX object** is created using the standard implementation of **IDispatch**, the type library that describes the object is loaded to provide the **IDispatch** implemenation.
- When the type information that interpreted the remoting code is asked to remote an interface across processes or across machines, the type library and type information that describes the interface must be loaded.
- When VBA 5.0 compiles an application that accesses objects or DLLs described by a type library, VBA 5.0 loads the type library and all type information in the type library to bind variable names.
- VBA 5.0 uses the standard implementation of **IDispatch** on its own classes. VBA 5.0 loads the type library and type information that describe the classes when the **IDispatch** pointer to a class that was created by VBA 5.0 is first handed out.
- When a user brings up the Find dialog on the object browser to find a member that contains a specific string, the object browser will load all references, type libraries, and all type information in the type libraries to get all of the available names.

Automation type library performance improvements come from two categories.

1. New file format and in-memory structure that enables the level information of the type library and each type information to be loaded in a single seek and read from disk with minimal additional initialization.
2. API-level improvements allow applications to access the data they need more efficiently. This includes features such as application-specific data and case-senstive identifiers.

VBA 5.0 Custom Data Storage

Custom data can be stored in a type library.

Users can store custom data in the form of (*guid, value*) pairs for each item in the type library (for example, the type library, type information, FUNCDESC, VARDESC). There can be any number of (*guid, value*) pairs for each item. The stored value can be any simple variant type, as well as arrays (not objects)of those types. If a GUID is to be stored as the value, it should be stored in ASCII format (as a BSTR) or in binary format (as an VT_ARRAY | VT_UI1), because GUID is not a simple variant type.

Case-Sensitive Identifiers

To provide support for case-sensitive identifier names in type libraries, the following changes in Automation have been made:

- The **/noi** switch for the MkTypLib utility (consistent with the Microsoft Linker). By default, type libraries are not case sensitive, unless this option is selected. The internal comparisons for MkTypLib are case-sensitive if this flag is set.
- **CreateTypeLib2** takes a flag to control case sensitivity (in addition to creating type libraries in the new file format).
- A new LIBFLAG_FCASESENSITIVE bit is added to the LIBFLAGS word of the TLIBATTR.

When a host application references a type library with multiple public names that differ only by case, the conflict is handled by the same code that resolves ambiguities between public names in different type libraries. The first one is selected.

Changes to Existing Data Structures

The only changed structure is the IDLDESC (now called PARAMDESC). The *dwReserved* field has been replaced by an *lpVarValue* field, which contains a pointer to a VARIANT describing the default value for this parameter (if the PARAMFLAG_FOPT and PARAMFLAG_FHASDEFAULT bit of *wParamFlags* has been set). For more information, refer to Chapter 6, "Data Types, Structures, and Enumerations."

The existing methods that return these data structures (**GetLibAttr, GetTypeAttr, GetFuncDesc,** and **GetVarDesc**) store the data that they return in a single, contiguous memory block allocated out of a local cache, rather than using **IMalloc**.

File Formats

The file format and in-memory format for Automation makes use of memory-mapped files for a 32-bit format and files used by the Apple Macintosh. The on-disk format matches the in-memory format, and matches the format in which the data is delivered to the user. The on-disk format is arranged into sections, so that the data that is likely to be read together is stored together.

Creating a New TypeLib

The **CreateTypeLib2()** API creates a type library in the new file format. The existing **CreateTypeLib()** API continues to create a type library in the previous format (using the previous code). When Visual Basic version 4.0 makes an executable file, the type library that is created and stored as part of the this file is in the previous file format.

The files MkTyplib.exe and Midl.exe, as well as the VBA 5.0 code, use the new **CreateTypeLib2()** API.

Type Building Interfaces

The type building interfaces, **ICreateTypeInfo** and **ICreateTypeLib**, are used to build tools that automate the process of generating type descriptions and creating type libraries. The MkTypLib utility and the MIDL compiler, for example, use these interfaces to create type libraries. For more information about type libraries, refer to Chapter 8, "Type Libraries and the Object Description Language."

Generally, it is not necessary to write custom implementations of these interfaces. The compilers use the default implementations that are returned by the **CreateTypeLib** function. To create tools similar to MkTypLib, the default implementations can be called.

Implemented by	Used by	Header file name	Import library name
Oleaut32.dll (32-bit systems) Typelib.dll (16-bit systems)	Applications that expose programmable objects.	Oleauto.h Dispatch.h	Oleaut32.lib Typelib.lib

Overview of Type Building Interfaces

The type building interfaces include the following member functions:

Interface	Member function	Purpose
ICreateTypeInfo	**AddFuncDesc**	Adds a function description as a type description.
	AddImplType	Specifies an inherited interface.
	AddRefTypeInfo	Adds a type description to those referenced by the type description being created.
	AddVarDesc	Adds a data member description as a type description.
	DefineFuncAsDllEntry	Associates a DLL entry point with a function that has a specified index.
ICreateTypeInfo	**LayOut**	Assigns VTBL offsets for virtual functions and instance offsets for per-instance data members.
	SetAlignment	Specifies data alignment for types of TKIND_RECORD.
	SetDocString	Sets the documentation string displayed by type browsers.
	SetFuncAndParamNames	Sets the function name and names of its

		parameters.
	SetFuncDocString	Sets the documentation string for a function.
	SetFuncHelpContext	Sets the Help context ID for a function.
	SetGuid	Sets the globally unique ID for the type library.
	SetHelpContext	Sets the Help context ID of the type description.
	SetImplTypeFlags	Sets the attributes for an implemented or inherited interface of a type.
	SetMops	Sets the *opcode* string for a type description.
	SetSchema	Reserved for future use.
	SetTypeDescAlias	Sets the type description for which this type description is an alias, if TYPEKIND=TKIND_ALIAS.
	SetTypeFlags	Sets type flags of the type description that is being created.
	SetTypeIdlDesc	Reserved for future use.
	SetVarDocString	Sets the documentation string for a variable.
	SetVarHelpContext	Sets the Help context ID for a variable.
	SetVarName	Sets the name of a variable.
	SetVersion	Sets version numbers for the type description.
ICreateTypeLib	**CreateTypeInfo**	Creates a new type description instance within the type library.
	SaveAllChanges	Saves the **ICreateTypeLib** instance.
	SetDocString	Sets the documentation string for the type library.
	SetGuid	Sets the globally unique ID for the type library.
	SetHelpContext	Sets the Help context ID for general information about the type library in the Help file.
	SetHelpFileName	Sets the Help file name.
	SetLcid	Sets the locale ID code indicating the national language associated with the library.

	SetLibFlags	Sets library flags, such as LIBFLAG_FRESTRICTED.
	SetName	Sets the name of the type library.
	SetVersion	Sets major and minor version numbers for the type library.
ICreateTypeInfo2		The **ICreateTypeInfo** instance returned from **ICreateTypeLib** can be accessed through a **QueryInterface()** call to **ICreateTypeInfo2**.
ICreateTypeLib2		Inherits from **ICreateTypeLib** and adds a method that supports removing type information from a library.
Library creation functions	**CreateTypeLib**	Gives access to a new object instance that supports the **ICreateTypeLib** interface.

You create an Automation type library by using the **ICreateTypeLib** and **ICreateTypeInfo** interfaces.

In the following example, a type library is created (Hello.tlb) by the MIDL compiler (or MkTyplib.exe), using the following .odl file.

```
 [
uuid(2F6CA420-C641-101A-B826-00DD01103DE1),            // LIBID_Hello
helpstring("Hello 1.0 Type Library"),
lcid(0x0409),
version(1.0)
]
library Hello
{
#ifdef WIN32
importlib("stdole32.tlb");
#else
importlib("stdole.tlb");
#endif

[
uuid(2F6CA422-C641-101A-B826-00DD01103DE1),        // IID_IHello
helpstring("Hello Interface")
]
interface IHello : IUnknown
{
[propput] void HelloMessage([in] BSTR Message);
[propget] BSTR HelloMessage(void);
void SayHello(void);
}
[
```

```
uuid(2F6CA423-C641-101A-B826-00DD01103DE1),        // IID_DHello
helpstring("Hello Dispinterface")
]
dispinterface DHello
{
interface IHello;
}

[
uuid(2F6CA421-C641-101A-B826-00DD01103DE1),           // CLSID_Hello.
helpstring("Hello Class")
]
coclass Hello
{
dispinterface DHello;
interface IHello;
}
}
```

ICreateTypeInfo Interface

The **ICreateTypeInfo** interface provides the tools for creating and administering the type information defined through the type description.

ICreateTypeInfo::AddFuncDesc

HRESULT ICreateTypeInfo::AddFuncDesc(*index*, *lpFuncDesc***)**
unsigned int *index*
FUNCDESC FAR* *lpFuncDesc*

Adds a function description to the type description.

Parameters

index

Index of the new FUNCDESC in the type information.

lpFuncDesc

Pointer to a FUNCDESC structure that describes the function. The *bstrIDLInfo* field in the FUNCDESC should be set to Null for future compatibility.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

The index specifies the order of the functions within the type information. The first function has an index of zero. If an index is specified that exceeds one less than the number of functions in the type information, an error is returned. Calling this function does not pass ownership of the FUNCDESC structure to **ICreateTypeInfo**. Therefore, the caller must still de-allocate the FUNCDESC structure.

The passed-in VTBL field (*oVft*) of the FUNCDESC is ignored. This attribute is set when **ICreateTypeInfo::LayOut** is called.

The function **AddFuncDesc** uses the passed-in member ID fields within each FUNCDESC for classes with TYPEKIND = TKIND_DISPATCH or TKIND_INTERFACE. If the member IDs are set to MEMBERID_NIL, **AddFuncDesc** assigns member IDs to the functions. Otherwise, the member ID fields within each FUNCDESC are ignored.

Any HREFTYPE fields in the FUNCDESC structure must have been produced by the same instance of **ITypeInfo** for which **AddFuncDesc** is called.

The **get** and **put** accessor functions for the same property must have the same dispatch ID (DISPID).

ICreateTypeInfo::AddImplType

HRESULT ICreateTypeInfo::AddImplType(*index, hreftype***)**
unsigned int *index*
HREFTYPE *hreftype*

Specifies an inherited interface, or an interface implemented by a component object class (coclass).

Parameters

index

Index of the implementation class to be added. Specifies the order of the type relative to the other type.

hreftype

Handle to the referenced type description obtained from the **AddRefType** description.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

To specify an inherited interface, use *index* = 0. For a dispinterface with Syntax 2, call **ICreateTypeInfo::AddImplType** twice, once with *nindex* = 0 for the inherited **IDispatch** and once with *nindex* = 1 for the interface that is being wrapped. For a dual interface, call **ICreateTypeInfo::AddImplType** with
nindex = -1 for the TKIND_INTERFACE type information component of the dual interface.

ICreateTypeInfo::AddRefTypeInfo

HRESULT ICreateTypeInfo::AddRefTypeInfo(*lptinfo*, *lphreftype***)**
ITypeInfo FAR* *lptinfo*
HREFTYPE FAR* *lphreftype*

Adds a type description to those referenced by the type description being created.

Parameters

lptinfo

Pointer to the type description to be referenced.

lphreftype

On return, pointer to the handle that this type description associates with the referenced type information.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

The second parameter returns a pointer to the handle of the added type information. If **AddRefTypeInfo** has been called previously for the same type information, the index that was returned by the previous call is returned in *lphreftype*. If the referenced type description is in the type library being created, its type information can be obtained by calling **IUnknown::QueryInterface(**IID_ITypeInfo, ...**)** on the **ICreateTypeInfo** interface of that type description.

ICreateTypeInfo::AddVarDesc

HRESULT ICreateTypeInfo::AddVarDesc(*index*, *lpVarDesc***)**
unsigned int *index*
VARDESC FAR* *lpVarDesc*

Adds a variable or data member description to the type description.

Parameters

index

Index of the variable or data member to be added to the type description.

lpVarDesc

Pointer to the variable or data member description to be added.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

The index specifies the order of the variables. The first variable has an index of zero. **ICreateTypeInfo::AddVarDesc** returns an error if the specified index is greater than the number of variables currently in the type information. Calling this function does not pass ownership of the VARDESC structure to **ICreateTypeInfo**. The instance field (*oInst*) of the VARDESC structure is ignored. This attribute is set only when **ICreateTypeInfo::LayOut** is called. Also, the member ID fields within the VARDESCs are ignored unless the TYPEKIND of the class is TKIND_DISPATCH.

Any HREFTYPE fields in the VARDESC structure must have been produced by the same instance of **ITypeInfo** for which **AddVarDesc** is called.

AddVarDesc ignores the contents of the *idldesc* field of the ELEMDESC.

ICreateTypeInfo::DefineFuncAsDllEntry

HRESULT ICreateTypeInfo::DefineFuncAsDllEntry(*index*, *szDllName*, *szProcName***)**
unsigned int *index*
OLECHAR FAR* *szDllName*
OLECHAR FAR* *szProcName*

Associates a DLL entry point with the function that has the specified *index*.

Parameters

index

Index of the function.

szDllName

Name of the DLL that contains the entry point.

szProcName

Name of the entry point or an ordinal (if the high word is zero).

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_ELEMENTNOTFOUND	The element cannot be found.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

If the high word of *szProcName* is zero, then the low word must contain the ordinal of the entry point; otherwise, *szProcName* points to the zero-terminated name of the entry point.

ICreateTypeInfo::LayOut

HRESULT ICreateTypeInfo::LayOut()

Assigns VTBL offsets for virtual functions and instance offsets for per-instance data members, and creates the two type descriptions for dual interfaces.

Parameters

None.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_UNDEFINEDTYPE	Bound to unrecognized type.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.
TYPE_E_WRONGTYPEKIND	Type mismatch.
TYPE_E_ELEMENTNOTFOUND	The element cannot be found.
TYPE_E_AMBIGUOUSNAME	More than one item exists with this name.
TYPE_E_SIZETOOBIG	The type information is too long.
TYPE_E_TYPEMISMATCH	Type mismatch.

Comments

LayOut also assigns member ID numbers to the functions and variables, unless the TYPEKIND of the class is TKIND_DISPATCH. Call **LayOut** after all members of the type information are defined, and before the type library is saved.

Use **ICreateTypeLib::SaveAllChanges** to save the type information after calling **LayOut**. Other members of the **ICreateTypeInfo** interface should not be called after calling **LayOut**.

Note Different implementations of **ICreateTypeInfo** or other interfaces that create type information are free to assign any member ID numbers, provided that all members (including inherited members), have unique IDs. For examples, see the **ICreateTypeInfo2** interface later in this chapter..

ICreateTypeInfo::SetAlignment

HRESULT ICreateTypeInfo::SetAlignment(*cbAlignment***)**
unsigned short *cbAlignment*

Specifies the data alignment for an item of TYPEKIND=TKIND_RECORD.

Parameter

cbAlignment

Alignment method for the type. A value of 0 indicates alignment on the 64K boundary; 1 indicates no special alignment. For other values, *n* indicates alignment on byte *n*.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Comments

The alignment is the minimum of the natural alignment (for example, byte data on byte boundaries, word data on word boundaries, and so on), and the alignment denoted by *cbAlignment*.

ICreateTypeInfo::SetDocString

HRESULT ICreateTypeInfo::SetDocString(*szDoc***)**
OLECHAR FAR* *szDoc*

Sets the documentation string displayed by type browsers.

Parameter

szDoc

Pointer to the documentation string.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Comments

The documentation string is a brief description of the type description being created.

ICreateTypeInfo::SetFuncAndParamNames

HRESULT ICreateTypeInfo::SetFuncAndParamNames(*index*, *rgszNames*, *cNames***)**
unsigned int *index*
OLECHAR FAR* FAR* *rgszNames*
unsigned int *cNames*

Sets the name of a function and the names of its parameters to the names in the array of pointers *rgszNames*.

Parameters

index

Index of the function whose function name and parameter names are to be set.

rgszNames

Array of pointers to names. The first element is the function name. Subsequent elements are names of parameters.

cNames

Number of elements in the *rgszNames* array.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_ELEMENTNOTFOUND	The element cannot be found.

Comments

The function **SetFuncAndParamNames** needs to be used once for each property. The last parameter for **put** and **putref** accessor functions is unnamed.

ICreateTypeInfo::SetFuncDocString

HRESULT ICreateTypeInfo::SetFuncDocString(*index*, *szDocString***)**
unsigned int *index*
OLECHAR FAR* *szDocString*

Sets the documentation string for the function with the specified *index*.

Parameters

index

Index of the function.

szDocString

Pointer to the documentation string.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_ELEMENTNOTFOUND	The element cannot be found.

Comments

The documentation string is a brief description of the function intended for use by tools such as type browsers. **SetFuncDocString** only needs to be used once for each property, because all property accessor functions are identified by one name.

ICreateTypeInfo::SetFuncHelpContext

HRESULT ICreateTypeInfo::SetFuncHelpContext(*index, dwHelpContext***)**
unsigned int *index*
unsigned long *dwHelpContext*

Sets the Help context ID for the function with the specified *index*.

Parameters

index

Index of the function.

dwHelpContext

Help context ID for the Help topic.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_ELEMENTNOTFOUND	The element cannot be found.
E_INVALIDARG	One or more of the arguments is invalid.

Comments

SetFuncHelpContext only needs to be set once for each property, because all property accessor functions are identified by one name.

ICreateTypeInfo::SetGuid

HRESULT ICreateTypeInfo::SetGuid(*guid***)**
REFGUID *guid*

Sets the globally unique ID (GUID) associated with the type description.

Parameter

guid

Globally unique ID to be associated with the type description.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.

Comments

For an interface, this is an interface ID; for a coclass, it is a class ID. For information on GUIDs, see Chapter 8, "Type Libraries and the Object Description Language.'

ICreateTypeInfo::SetHelpContext

HRESULT ICreateTypeInfo::SetHelpContext(*dwHelpContext***)**
unsigned long *dwHelpContext*

Sets the Help context ID of the type information.

Parameter

dwHelpContext

Handle to the Help context.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.

ICreateTypeInfo::SetImplTypeFlags

HRESULT ICreateTypeInfo::SetImplTypeFlags(*index, impltypeflags***)**
unsigned int *index*
int *impltypeflags*

Sets the attributes for an implemented or inherited interface of a type.

Parameters

index

Index of the interface for which to set type flags.

impltypeflags

IMPLTYPE flags to be set.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.

Comments

SetImplTypeFlags sets the IMPLTYPE flags for the indexed interface. For more information, see the "IMPLTYPEFLAGS" section in Chapter 9, "Type Description Interfaces."

ICreateTypeInfo::SetMops

HRESULT ICreateTypeInfo::SetMops(*index, bstrMops***)**
unsigned int *index*
BSTR *bstrMops*

Sets the marshaling *opcode* string associated with the type description or the function.

Parameters

index

Index of the member for which to set the *opcode* string. If *index* is -1, sets the *opcode* string for the type description.

bstrMops

The marshaling *opcode* string.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.

ICreateTypeInfo::SetTypeDescAlias

HRESULT ICreateTypeInfo::SetTypeDescAlias(*lptDescAlias***)**
TYPEDESC FAR* *lptDescAlias*

Sets the type description for which this type description is an alias, if TYPEKIND=TKIND_ALIAS.

Parameter

lptDescAlias

Pointer to a type description that describes the type for which this is an alias.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

To set the type for an alias, call **SetTypeDescAlias** for a type description whose TYPEKIND is TKIND_ALIAS.

ICreateTypeInfo::SetTypeFlags

HRESULT ICreateTypeInfo::SetTypeFlags(*uTypeFlags***)**
unsigned int *uTypeFlags*

Sets type flags of the type description being created.

Parameter

uTypeFlags

Settings for the type flags.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

Use **SetTypeFlags** to set the flags for the type description. For details, see the "TYPEFLAGS" section in Chapter 9, "Type Description Interfaces."

ICreateTypeInfo::SetVarDocString

HRESULT ICreateTypeInfo::SetVarDocString(*index*, *szDocString***)**
unsigned int *index*
OLECHAR FAR* *szDocString*

Sets the documentation string for the variable with the specified *index*.

Parameters

index

Index of the variable being documented.

szDocString

The documentation string to be set.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_ELEMENTNOTFOUND	The element was not found.

ICreateTypeInfo::SetVarHelpContext

HRESULT ICreateTypeInfo::SetVarHelpContext(*index*, *dwHelpContext***)**
unsigned int *index*
unsigned long *dwHelpContext*

Sets the Help context ID for the variable with the specified *index*.

Parameters

index

Index of the variable described by the type description.

dwHelpContext

Handle to the Help context ID for the Help topic on the variable.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_ELEMENTNOTFOUND	The element cannot be found.

ICreateTypeInfo::SetVarName

HRESULT ICreateTypeInfo::SetVarName(*index*, *szName***)**
unsigned int *index*
OLECHAR FAR* *szName*

Sets the name of a variable.

Parameters

index

Index of the variable whose name is being set.

szName

Name for the variable.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_ELEMENTNOTFOUND	The element cannot be found.

ICreateTypeInfo::SetVersion

HRESULT ICreateTypeInfo::SetVersion(*wMajorVerNum*, *wMinorVerNum***)**
unsigned short *wMajorVerNum*
unsigned short *wMinorVerNum*

Sets the major and minor version number of the type information.

Parameters

wMajorVerNum

Major version number for the type.

wMinorVerNum

Minor version number for the type.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_ACCESSDENIED	Cannot write to the destination.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Library Creation Functions

The following API and interface methods support the creation and administration of type libraries and type descriptions.

CreateTypeLib Quick Info

HRESULT CreateTypeLib(*syskind*, *szFile*, *lplpctlib***)**
SYSKIND *syskind*
OLECHAR FAR* *szFile*
ICreateTypeLib FAR* FAR* *lplpctlib*

Provides access to a new object instance that supports the **ICreateTypeLib** interface.

Parameters

syskind

The target operating system for which to create a type library.

szFile

The name of the file to create.

lplpctlib

Pointer to an instance supporting the **ICreateTypeLib** interface.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function could not create the file.
Other return codes	All FACILITY_STORAGE errors.

Comments

CreateTypeLib sets its output parameter (*lplpctlib*) to point to a newly created object that supports the **ICreateTypeLib** interface.

ICreateTypeLib Interface

The **ICreateTypeLib** interface provides the methods for creating and managing the component or file that contains type information. Type libraries are created from type descriptions using the MkTyplib utility or the MIDL compiler. These type libraries are accessed through the **ITypeLib** interface.

ICreateTypeLib::CreateTypeInfo

HRESULT ICreateTypeLib::CreateTypeInfo(*szName*, *tkind*, *lplpctinfo***)**
OLECHAR FAR* *szName*
TYPEKIND *tkind*
ICreateTypeInfo FAR* FAR* *lplpctinfo*

Creates a new type description instance within the type library.

Parameters

szName

Name of the new type.

tkind

TYPEKIND of the type description to be created.

lplpctinfo

On return, contains a pointer to the type description.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.
TYPE_E_NAMECONFLICT	The provided name is not unique.
TYPE_E_WRONGTYPEKIND	Type mismatch.

Comments

use the function **CreateTypeInfo** to create a new type description instance within the library. An error is returned if the specified name already appears in the library. Valid *tkind* values are described in the "TYPEKIND" section in Chapter 9, "Type Description Interfaces." To get the type information of the type description that is being created, call **IUnknown::QueryInterface(**IID_ITypeInfo, ...**)** on the returned **ICreateTypeInfo**. This type information can be used by other type descriptions that reference it by using **ICreateTypeInfo::AddRefTypeInfo.**

ICreateTypeLib::SaveAllChanges

HRESULT ICreateTypeLib::SaveAllChanges()

Saves the **ICreateTypeLib** instance following the layout of type information.

Parameters

None.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function cannot write to the file.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.
Other return codes	All FACILITY_STORAGE errors.

Comments

You should not callany other **ICreateTypeLib** methods after calling **SaveAllChanges**.

ICreateTypeLib::SetDocString

HRESULT ICreateTypeLib::SetDocString(*szDoc***)**
OLECHAR FAR* *szDoc*

Sets the documentation string associated with the library.

Parameter

szDoc

A documentation string that briefly describes the type library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

Comments

The documentation string is a brief description of the library intended for use by type information browsing tools.

ICreateTypeLib::SetGuid

HRESULT ICreateTypeLib::SetGuid(*guid***)**
REFGUID *guid*

Sets the universal unique ID (UUID) associated with the type library (Also known as the globally unique identifier (GUID)).

Parameter

guid

The globally unique ID to be assigned to the library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Comments

Universal unique IDs (UUIDs) are described in Chapter 8, "Type Libraries and the Object Description Language."

ICreateTypeLib::SetHelpContext

HRESULT ICreateTypeLib::SetHelpContext(*dwHelpContext***)**
unsigned long *dwHelpContext*

Sets the Help context ID for retrieving general Help information for the type library.

Parameter

dwHelpContext

Help context ID to be assigned to the library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Comments

Calling **SetHelpContext** with a Help context of zero is equivalent to not calling it at all, because zero indicates a null Help context.

ICreateTypeLib::SetHelpFileName

HRESULT ICreateTypeLib::SetHelpFileName(*szFileName***)**
OLECHAR FAR* *szFileName*

Sets the name of the Help file.

Parameter

szFileName

The name of the Help file for the library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Comments

Each type library can reference a single Help file.

The **GetDocumentation** method of the created **ITypeLib** returns a fully qualified path for the Help file, which is formed by appending the name passed into *szFileName* to the registered Help directory for the type library. The Help directory is registered under:

\TYPELIB\<*guid of library*>\<*Major.Minor version* >\HELPDIR

ICreateTypeLib::SetLibFlags

HRESULT ICreateTypeLib::SetLibFlags(*uLibFlags***)**
unsigned int *uLibFlags*

Sets library flags, such as LIBFLAG_FRESTRICTED.

Parameter

uLibFlags

The flags to set for the library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Comments

Valid *uLibFlags* values are listed in "LIBFLAGS," in Chapter 6, "Data Types, Structures, and Enumerations."

ICreateTypeLib::SetLcid

HRESULT ICreateTypeLib::SetLcid(*lcid***)**
LCID *lcid*

Sets the binary Microsoft national language ID associated with the library.

Parameter

lcid

Represents the locale ID for the type library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

Comments

For more information on national language IDs, see "Supporting Multiple National Languages," in Chapter 2, "Exposing Automation Objects." For additional information for 16-bit systems, refer to Appendix A, "National Language Support Functions." For 32-bit systems, refer to Windows NT documentation on the National Language Support (NLS) API.

ICreateTypeLib::SetName

HRESULT ICreateTypeLib::SetName(*szName***)**
OLECHAR FAR* *szName*

Sets the name of the type library.

Parameter

szName

Name to be assigned to the library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
STG_E_INSUFFICIENTMEMORY	Out of memory.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

ICreateTypeLib::SetVersion

HRESULT SetVersion(*wMajorVerNum*, *wMinorVerNum***)**
unsigned short *wMajorVerNum*
unsigned short *wMinorVerNum*

Sets the major and minor version numbers of the type library.

Parameters

wMajorVerNum

Major version number for the library.

wMinorVerNum

Minor version number for the library.

Return Value

The return value of the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
TYPE_E_INVALIDSTATE	The state of the type library is not valid for this operation.

CreateTypeLib2 API

The **CreateTypeLib2** API creates a type library in the current file format.

The file and in-memory format for this current version of Automation makes use of memory-mapped files for 32-bit (and files for the Apple Macintosh). The existing **CreateTypeLib()** API is still available for creating a type library in the older format.

HRESULT CreateTypeLib2(*syskind, lpszFileName, ppctlib***)**
SYSKIND *syskind*
LPOLESTR *lpszFileName*
ICreateTypeLib** *ppctlib*

ICreateTypeLib2 Interface

ICreateTypeLib2 inherits from **ICreateTypeLib,** and adds three methods that support removing a type information from a library. The **ICreateTypeInfo** instance returned from **ICreateTypeLib** can be accessed through a **QueryInterface()** call to **ICreateTypeInfo2**.

Example

```
interface ICreateTypeLib2 : ICreateTypeLib
```

ICreateTypeLib2::DeleteTypeInfo

HRESULT ICreateTypeLib2::DeleteTypeInfo(*szName***)**;
OLECHAR **szName*

Deletes a specified type information from the type library.

Parameter

szName

Name of the type information to remove.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeLib2::SetCustData

HRESULT ICreateTypeLib2::SetCustData(*rguid, pVarVal***)**
REFGUID r*guid*
VARIANT ******pVarVal*

Sets a value to custom data.

Parameter

rguid

Unique identifier used to identify the data.

pVarVal

The data to store (any variant except an object).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeLib2::SetHelpStringContext

HRESULT ICreateTypeLib2::SetHelpStringContext(*dwHelpStringContext***)**
DWORD ***dwHelpStringContext**

Sets the Help string context number.

Parameter

DwHelpStringContext

The Help string context number.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeLib2::SetHelpStringDll

HRESULT ICreateTypeLib2::SetHelpStringDll(*szFileName***)**
LPOLESTR *szFileName*

Sets the DLL name to be used for Help string lookup (for localization purposes).

Parameter

szFileName

The DLL file name.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2 Interface

The **ICreateTypeInfo2** interface derives from **ICreateTypeInfo,** and adds methods for deleting items that have been added through **ICreateTypeInfo**.

The **ICreateTypeInfo::LayOut** method provides a way for the creator of the type information to check for any errors. A call to **QueyInterface()** can be made to the **ICreateTypeInfo** instance at any time for its **ITypeInfo** interface. Calling any of the methods in the **ITypeInfo** interface that require layout information lays out the type information automatically.

Example

```
interface ICreateTypeInfo2 : ICreateTypeInfo
```

ICreateTypeInfo2::DeleteFuncDesc

HRESULT ICreateTypeInfo2::DeleteFuncDesc (*index***)**
unsigned int *index*

Deletes a function description specified by the index number.

Parameter

index

Index of the function whose description is to be deleted. The index should be in the range of 0 to 1 less than the number of functions in this type.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::DeleteFuncDescByMemId

HRESULT ICreateTypeInfo2::DeleteFuncDescByMemId(*memid, invkind***)**
MEMBERID *memid*
INVOKEKIND *invkind*

Deletes the function description (FUNCDESC) specified by *memid*.

Parameters

memid

Member identifier of the FUNCDESC to delete.

invkind

The type of the invocation.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::DeleteVarDesc

HRESULT ICreateTypeInfo2::DeleteVarDesc(*index***)**
unsigned int *index*

Deletes the specified VARDESC structure.

Parameter

index

Index number of the VARDESC structure.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function cannot read from the file.
TYPE_E_INVDATAREAD	The function cannot read from the file.
TYPE_E_UNSUPFORMAT	The type library has an old format.
TYPE_E_INVALIDSTATE	The type library cannot be opened.

Example

```
ptypeinfo->DeleteVarDesc(index);
```

ICreateTypeInfo2::DeleteVarDescByMemId

HRESULT ICreateTypeInfo2::DeleteVarDescByMemId(*memid***)**
MEMBERID *memid*

Deletes the specified VARDESC structure.

Parameter

memid

Member identifier of the VARDESC to be deleted.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.
TYPE_E_IOERROR	The function cannot read from the file.
TYPE_E_INVDATAREAD	The function cannot read from the file.
TYPE_E_UNSUPFORMAT	The type library has an older format.
TYPE_E_INVALIDSTATE	The type library cannot be opened.

ICreateTypeInfo2::DeleteImplType

HRESULT ICreateTypeInfo2::DeleteImplType(*index***)**
unsigned int *index*

Deletes the IMPLTYPE flags for the indexed interface.

Parameter

index

Index of the interface for which to delete the type flags.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::SetCustData

HRESULT ICreateTypeInfo2::SetCustData(*rguid, pVarVal***)**
REFGUID r*guid*
VARIANT ***pVarVal*

Sets a value for custom data.

Parameter

rguid

Unique identifier that can be used to identify the data.

pVarVal

The data to store (any variant except an object).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::SetHelpStringContext

HRESULT SetHelpStringContext(*pdwHelpStringContext***)**
DWORD **pdwHelpStringContext*

Sets the context number for the specified Help string.

Parameter

pdwHelpStringContext

Pointer to the Help string context number.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	Argument is invalid.

ICreateTypeInfo2::SetFuncCustData

HRESULT SetFuncCustData(*index, guid, pVarVal***)**
unsigned int *index*,
REFGUID *guid*
VARIANT **pVarVal*

Sets a value for a specified custom function.

Parameter

index

The index of the function for which to set the custom data.

rguid

Unique identifier used to identify the data.

pVarVal

The data to store (any variant except an object).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::SetFuncHelpStringContext

HRESULT SetFuncHelpStringContext(*index, dwHelpStringContext***)**
unsigned int *index*,
DWORD *dwHelpStringContext*

Sets a Help context value for a specified custom function.

Parameter

index

The index of the function for which to set the custom data.

dwHelpStringContext

Help string context for a localized string

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::SetVarCustData

HRESULT SetVarCustData(*index*, *guid, pVarVal***)**
unsigned int *index*
REFGUID *guid*
VARIANT **pVarVal*

Sets a custom data variable.

Parameter

index

Index of the variable for which to set the custom data.

guid

Globally unique ID (GUID) used to identify the data.

pVarVal

Data to store (any legal variant except an object).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::SetParamCustData

HRESULT SetParamCustData(*indexFunc, indexParam*, *guid, pVarVal***)**
unsigned int *indexFunc*
unsigned int *indexParam*
REFGUID *guid*
VARIANT **pVarVal*

Sets the specified parameter for the custom data.

Parameter

indexFunc

Index of the function for which to set the custom data.

indexParam

Index of the parameter of the function for which to set the custom data.

guid

Globally unique ID (GUID) used to identify the data.

pvarVal

The data to store (any legal variant except an object).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::SetImplTypeCustData

HRESULT SetImplTypeCustData(*index, guid, pVarVal* **)**
unsigned int *index*
REFGUID *guid*
VARIANT **pVarVal*

Sets the implementation type for custom data.

Parameter

index

Index of the variable for which to set the custom data.

guid

Unique identifier used to identify the data.

pVarVal

Reference to the value of the variable.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ICreateTypeInfo2::SetVarHelpStringContext

HRESULT SetVarHelpStringContext(*index, dwHelpStringContext***)**
unsigned int *index*,
DWORD *dwHelpStringContext*

Sets a Help context value for a specified variable.

Parameter

index

The index of the variable.

dwHelpStringContext

Help string context for a localized string

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeLib2 Interface

The **ITypeLib2** interface inherits from **ITypeLib**. This allows an **ITypeLib** to be cast to an **ITypeLib2** in performance-sensitive cases, rather than performing an extra **QueryInterface()** and **Release()**.

```
DECLARE_INTERFACE_(ITypeLib2, ITypeLib)
{
BEGIN_INTERFACE
```

ITypeLib2::GetCustData

HRESULT GetCustData(*guid, pVarVal***)**
REFGUID *guid*
VARIANT **pVarVal*

Gets the custom data.

Parameter

guid

Globally unique ID (GUID) used to identify the data.

pVarVal

The location of where to put the retrieved data.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeLib2::GetLibStatistics

HRESULT GetLibStatistics(*pcUniqueNames, pcchUniqueNames***)**
unsigned long **pcUniqueNames*
unsigned long **pcchUniqueNames*

Returns type library statistics that are required for efficient sizing of hash tables.

Parameter

pcUniqueNames

Unique name strings

pcchUniqueNames

Character count of return strings

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

ITypeLib2::GetHelpStringContext

HRESULT GetHelpStringContext(*index, pdwHelpStringContext***)**
unsigned int *index*
unsigned long **pdwHelpStringContext*

Gets the context number for the specified Help string.

Parameter

index

Index of the Help string.

pdwHelpStringContext

Help string context number (DWORD).

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Out of memory.
E_INVALIDARG	One or more of the arguments is invalid.

Error Handling Interfaces

Objects that are invoked through VTBL binding need to use the Automation error handling interfaces to define and return error information. The interfaces include the following:

- **ICreateErrorInfo** – Sets error information.
- **IErrorInfo** – Returns information from an error object.
- **ISupportErrorInfo** – Identifies this object as supporting the **IErrorInfo** interface.
- Error handling API functions.

This chapter covers the error handling interfaces. The member functions of each interface are listed in the following table.

Category	Member function	Purpose
IErrorInfo	**GetDescription**	Returns a textual description of the error.
	GetGUID	Returns the globally unique ID for the interface that defined the error.
	GetHelpContext	Returns the Help context ID for the error.
	GetHelpFile	Returns the path of the Help file that describes the error.
	GetSource	Returns the ProgID for the class or application that returned the error.
ICreateErrorInfo	**SetDescription**	Sets a textual description of the error.
	SetGUID	Sets the globally unique ID for the interface that defined the error.
	SetHelpContext	Sets the Help context ID for the error.
	SetHelpFile	Sets the path of the Help file that describes the error.
	SetSource	Sets the programmatic identifier (ProgID) for the class or application that returned the error.
ISupportErrorInfo	**InterfaceSupportsErrorInfo**	Indicates whether an interface supports the **IErrorInfo** interface.
Error handling functions	CreateErrorInfo	Creates a generic error object.
	GetErrorInfo	Retrieves and clears the current error object.
	SetErrorInfo	Sets the current error

object.

Returning Error Information

Quick Info

To return error information

1. Implement the **ISupportErrorInfo** interface.
2. To create an instance of the generic error object, call the **CreateErrorInfo** function.
3. To set its contents, use the **ICreateErrorInfo** methods.
4. To associate the error object with the current logical thread, call the **SetErrorInfo** function.

The following figure illustrates this procedure.

{ewc msdncd, EWGraphic, bsd23530 0 /a "SDK_01.WMF"}

The error handling interfaces create and manage an error object, which provides information about the error. The error object is not the same as the object that encountered the error. It is a separate object associated with the current thread of execution.

Retrieving Error Information

Quick Info

To retrieve error information

1. Check whether the returned value represents an error that the object is prepared to handle.
2. Call **QueryInterface** to get a pointer to the **ISupportErrorInfo** interface. Then, call **InterfaceSupportsErrorInfo** to verify that the error was raised by the object that returned it and that the error object pertains to the current error, and not to a previous call.
3. To get a pointer to the error object, call the **GetErrorInfo** function.
4. To retrieve information from the error object, use the **IErrorInfo** methods.

The following figure illustrates this procedure.

{ewc msdncd, EWGraphic, bsd23530 1 /a "SDK_02.WMF"}

If the object is not prepared to handle the error, but needs to propagate the error information further down the call chain, it should simply pass the return value to its caller. Because the **GetErrorInfo** function clears the error information and passes ownership of the error object to the caller, the function should be called only by the object that handles the error.

IErrorInfo Interface

The **IErrorInfo** interface provides detailed contextual error information.

Implemented by	Used by	Header filename	Import library name
Oleaut32.dll (32-bit systems) Ole2disp.dll (16-bit systems)	Applications that receive rich information.	Oleauto.h Dispatch.h	Oleaut32.lib Oledisp.lib

IErrorInfo::GetDescription

HRESULT IErrorInfo::GetDescription(*pbstrDescription***)**
BSTR ***pbstrDescription***

Returns a textual description of the error.

Parameter

pbstrDescription

Pointer to a brief string that describes the error.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.

Comments

The text is returned in the language specified by the locale ID (LCID) that was passed to **IDispatch::Invoke** for the method that encountered the error.

IErrorInfo::GetGUID

HRESULT IErrorInfo::GetGUID(*pguid***)**
GUID ***pguid**

Returns the globally unique ID (GUID) of the interface that defined the error.

Parameter

pguid

Pointer to a GUID, or GUID_NULL, if the error was defined by the operating system.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.

Comments

IErrorInfo::GetGUID returns the GUID of the interface that defined the error. If the error was defined by the system, **IErrorInfo::GetGUID** returns GUID_NULL.

This GUID does not necessarily represent the source of the error. The source is the class or application that raised the error. Using the GUID, an application can handle errors in an interface, independent of the class that implements the interface.

IErrorInfo::GetHelpContext

HRESULT IErrorInfo::GetHelpContext(*pdwHelpContext***)**
DWORD ***pdwHelpContext***

Returns the Help context ID for the error.

Parameter

pdwHelpContext

Pointer to the Help context ID for the error.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.

Comments

IErrorInfo::GetHelpContext returns the Help context ID for the error. To find the Help file to which it applies, use **IErrorInfo::GetHelpFile**.

IErrorInfo::GetHelpFile

HRESULT IErrorInfo::GetHelpFile(*pbstrHelpFile***)**
BSTR ***pbstrHelpFile***

Returns the path of the Help file that describes the error.

Parameter

pbstrHelpFile

Pointer to a string that contains the fully qualified path of the Help file.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.

Comments

IErrorInfo::GetHelpFile returns the fully qualified path of the Help file that describes the current error. **IErrorInfo::GetHelpContext** should be used to find the Help context ID for the error in the Help file.

IErrorInfo::GetSource

HRESULT IErrorInfo::GetSource(*pbstrSource***)**
BSTR ***pbstrSource***

Returns the language-dependent programmatic ID (ProgID) for the class or application that raised the error.

Parameter

pbstrSource

Pointer to a string containing a ProgID, in the form *progname.objectname*.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.

Comments

Use **IErrorInfo::GetSource** to determine the class or application that is the source of the error. The language for the returned ProgID depends on the locale ID (LCID) that was passed into the method at the time of invocation.

ICreateErrorInfo Interface

The **ICreateErrorInfo** interface returns error information.

Implemented by	Used by	Header filename	Import library name
Oleaut32.dll (32-bit systems) Oledisp.dll (16-bit systems)	Applications that return rich error information.	Oleauto.h Dispatch.h	Oleaut32.lib Oledisp.lib

ICreateErrorInfo::SetDescription

HRESULT ICreateErrorInfo::SetDescription(*szDescription***)**
LPCOLESTR ***szDescription***

Sets the textual description of the error.

Parameter

szDescription

A brief, zero-terminated string that describes the error.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

Comments

The text should be supplied in the language specified by the locale ID (LCID) that was passed to the method raising the error. For more information, see "LCID Attribute" in Chapter 8, "Type Libraries and the Object Description Language."

Example

```
hr = CreateErrorInfo(&pcerrinfo);
if (m_excepinfo.bstrDescription)
     pcerrinfo->SetDescription(m_excepinfo.bstrDescription);
```

ICreateErrorInfo::SetGUID

HRESULT ICreateErrorInfo::SetGUID(*rguid***)**
REFGUID *rguid*

Sets the globally unique ID (GUID) of the interface that defined the error.

Parameters

rguid

The GUID of the interface that defined the error, or GUID_NULL if the error was defined by the operating system.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

Comments

ICreateErrorInfo::SetGUID sets the GUID of the interface that defined the error. If the error was defined by the system, set **ICreateErrorInfo::SetGUID** to GUID_NULL.

This GUID does not necessarily represent the source of the error, however, the source is the class or application that raised the error. Using the GUID, applications can handle errors in an interface, independent of the class that implements the interface.

Example

```
hr = CreateErrorInfo(&pcerrinfo);
pcerrinfo->SetGUID(IID_IHello);
```

ICreateErrorInfo::SetHelpContext

HRESULT ICreateErrorInfo::SetHelpContext(*dwHelpContext***)**
DWORD *dwHelpContext*

Sets the Help context ID for the error.

Parameters

dwHelpContext

The Help context ID for the error.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

Comments

ICreateErrorInfo::SetHelpContext sets the Help context ID for the error. To establish the Help file to which it applies, use **ICreateErrorInfo::SetHelpFile**.

Example

```
hr = CreateErrorInfo(&pcerrinfo);
pcerrinfo->SetHelpContext(dwhelpcontext);
```

ICreateErrorInfo::SetHelpFile

HRESULT ICreateErrorInfo::SetHelpFile(*szHelpFile***)**
LPCOLESTR *szHelpFile*

Sets the path of the Help file that describes the error.

Parameter

szHelpFile

The fully qualified path of the Help file that describes the error.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

Comments

ICreateErrorInfo::SetHelpFile sets the fully qualified path of the Help file that describes the current error. Use **ICreateErrorInfo::SetHelpContext** to set the Help context ID for the error in the Help file.

Example

```
hr = CreateErrorInfo(&pcerrinfo);
pcerrinfo->SetHelpFile("C:\myapp\myapp.hlp");
```

ICreateErrorInfo::SetSource

HRESULT ICreateErrorInfo::SetSource(*szSource***)**
LPCOLESTR *szSource*

Sets the language-dependent programmatic ID (ProgID) for the class or application that raised the error.

Parameter

szSource

A programmatic ID in the form *progname.objectname.*

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

Comments

ICreateErrorInfo::SetSource should be used to identify the class or application that is the source of the error. The language for the returned programmatic ID (ProgID) depends on the locale ID (LCID) that was passed to the method at the time of invocation.

Example

```
hr = CreateErrorInfo(&pcerrinfo);
if (m_excepinfo.bstrSource)
     pcerrinfo->SetSource(m_excepinfo.bstrSource);
```

ISupportErrorInfo Interface

The **ISupportErrorInfo** interface ensures that error information can be propagated up the call chain correctly. Automation objects that use the error handling interfaces must implement **ISupportErrorInfo.**

Implemented by	Used by	Header filename
Applications that return error information.	Applications that retrieve error information.	Oleauto.h (32-bit systems) Dispatch.h (16-bit systems)

ISupportErrorInfo::InterfaceSupportsErrorInfo

HRESULT ISupportErrorInfo::InterfaceSupportsErrorInfo(*riid*)
REFIID *riid*

Indicates whether or not an interface supports the **IErrorInfo** interface.

Parameter

riid

Pointer to an interface ID.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Interface supports **IErrorInfo.**
S_FALSE	Interface does not support **IErrorInfo**.

Comments

Objects that support the **IErrorInfo** interface must also implement this interface.

Programs that receive an error return value should call **QueryInterface** to get a pointer to the **ISupportErrorInfo** interface, and then call **InterfaceSupportsErrorInfo** with the *riid* of the interface that returned the return value. If **InterfaceSupportsErrorInfo** returns S_FALSE, then the error object does not represent an error returned from the caller, but from somewhere else. In this case, the error object can be considered incorrect and should be discarded.

If **ISupportErrorInfo** returns S_OK, use the **GetErrorInfo** function to get a pointer to the error object.

Example

The following example implements the **ISupportErrorInfo** for the Lines sample. The **IErrorInfo** implementation also supports the **AddRef**, **Release**, and **QueryInterface** members inherited from the **IUnknown** interface.

```
CSupportErrorInfo::CSupportErrorInfo(IUnknown FAR* punkObject, REFIID riid)
{
     m_punkObject = punkObject;
     m_iid = riid;
}


STDMETHODIMP
CSupportErrorInfo::QueryInterface(REFIID iid, void FAR* FAR* ppv)
{
     return m_punkObject->QueryInterface(iid, ppv);
}
```

```
STDMETHODIMP_(ULONG)
CSupportErrorInfo::AddRef(void)
{
     return m_punkObject->AddRef();
}

STDMETHODIMP_(ULONG)
CSupportErrorInfo::Release(void)
{
     return m_punkObject->Release();
}

STDMETHODIMP
CSupportErrorInfo::InterfaceSupportsErrorInfo(REFIID riid)
{
     return (riid == m_iid) ? NOERROR : ResultFromScode(S_FALSE);
}
```

Error Handling API Functions

For 32-bit systems, the error handling functions are provided in Oleaut32.dll, the header file is Oleauto.h, and the import library is Oleaut32.lib. For 16-bit systems, the error handling functions are provided in Ole2disp.dll, the header file is Dispatch.h, and the import library is Ole2disp.lib.

CreateErrorInfo Quick Info

HRESULT CreateErrorInfo(*ppcerrinfo***)**
ICreateErrorInfo ****ppcerrinfo***

Creates an instance of a generic error object.

Parameter

ppcerrinfo

Pointer to a system-implemented generic error object.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
E_OUTOFMEMORY	Could not create the error object.

Comments

This function returns a pointer to a generic error object, which you can use with **QueryInterface** on **ICreateErrorInfo** to set its contents. You can then pass the resulting object to **SetErrorInfo().** The generic error object implements both **ICreateErrorInfo** and **IErrorInfo**.

Example

```
ICreateErrorInfo *pcerrinfo;
HRESULT hr;

hr = CreateErrorInfo(&pcerrinfo);
```

GetErrorInfo Quick Info

HRESULT GetErrorInfo(, *dwReserved, pperrinfo***)**
DWORD *dwReserved*
IErrorInfo ***pperrinfo*

Obtains the error information pointer set by the previous call to **SetErrorInfo** in the current logical thread.

Parameter

dwReserved

Reserved for future use. Must be zero.

pperrinfo

Pointer to a pointer to an error object.

Return Value

The return value obtained from the returned HRESULT is one of the following:

Return value	Meaning
S_OK	Success.
S_FALSE	There was no error object to return.

Comments

This function returns a pointer to the most recently set **IErrorInfo** pointer in the current logical thread. It transfers ownership of the error object to the caller, and clears the error state for the thread.

SetErrorInfo Quick Info

HRESULT SetErrorInfo(*perrinfo***)**
DWORD *dwReserved*
IErrorInfo **perrinfo*

Sets the error information object for the current thread of execution.

Parameter

dwReserved

Reserved for future use. Must be zero.

perrinfo

Pointer to an error object.

Return Value

The return value obtained from the returned HRESULT is:

Return value	Meaning
S_OK	Success.

Comments

This function releases the existing error information object, if one exists, and sets the pointer to *perrinfo*. Use this function after creating an error object that associates the object with the current thread of execution.

If the property or method that calls **SetErrorInfo** is called by **DispInvoke,** then **DispInvoke** will fill the EXCEPINFO parameter with the values specified in the error information object. **DispInvoke** will return DISP_E_EXCEPTION when the property or method returns a failure return value for **DispInvoke.**

VTBL-binding controllers that do not use **IDispatch::Invoke** can get the error information object by using **GetErrorInfo**. This allows an object that supports a dualinterface to use **SetErrorInfo,** regardless of whether the client uses VTBL binding or **IDispatch**.

Example

```
ICreateErrorInfo *pcerrinfo;
     IErrorInfo *perrinfo;
     HRESULT hr;

hr = CreateErrorInfo(&pcerrinfo);
hr = pcerrinfo->QueryInterface(IID_IErrorInfo, (LPVOID FAR*) &perrinfo);
if (SUCCEEDED(hr))
     {
          SetErrorInfo(0, perrinfo);
          perrinfo->Release();
     }
     pcerrinfo->Release();
```

National Language Support Functions

The National Language Support (NLS) functions provide support for applications that use multiple locales at one time, especially applications that support Automation. Locale information is passed to allow the application to interpret both the member names and the argument data in the proper locale context. The information in this appendix applies only to 16-bit Windows systems. On 32-bit Windows systems, an NLS API is part of the system software.

Implemented by	Used by	Header filename	Import library name
Ole2nls.dll	Applications that support multiple national languages	Olenls.h	Ole2nls.lib

For Automation, applications need to get locale information and to compare and transform strings into the proper format for each locale.

A *locale* is simply user-preference information, represented as a list of values describing the user's language and sublanguage. National language support incorporates several disparate definitions of a locale into one coherent model. This model is designed to be general enough at a low level to support multiple, distinct, high-level functions, such as the ANSI C locale functions.

A *code page* is the mapping between character glyphs (shapes) and the 1-byte or 2-byte numeric values that are used to represent them. Microsoft Windows uses one of several code pages, depending on the installed localized version of Windows. For example, the Russian version uses code page 1251 (Cyrillic), while the English U.S. and Western European versions use code page 1252 (Multilingual). For historical reasons, the Windows code page in effect is referred to as the ANSI code page.

Because only one code page is in effect at a time, it is impossible for a computer running English U.S. Windows to display or print data correctly from the Cyrillic code page. The fonts do not contain the Cyrillic characters. However, it can still manipulate the characters internally, and they will display correctly again if moved back to a machine running Russian Windows.

All NLS functions use the locale ID (LCID) to identify which code page a piece of text is assumed to lie in. For example, when returning locale information (such as month names) for Russian, the returned string can be meaningfully displayed in the Cyrillic code page only, because other code pages do not contain the appropriate characters. Similarly, when attempting to change the case of a string with the Russian locale, the case-mapping rules assume the characters are in the Cyrillic code page.

These functions can be divided into two categories.

- String transformation – NLS functions support uppercasing, lowercasing, generating sort keys (all locale-dependent), and getting string type information.
- Locale manipulation – NLS functions return information about installed locales for use in string transformations.

Overview of Functions

The following table lists the NLS functions.

Function	Purpose
CompareStringA	Compares two strings of the same locale.
LCMapStringA	Transforms the case or sort order of a string.
GetLocaleInfoA	Retrieves locale information from the user's system.
GetStringTypeA	Retrieves locale type information about each character in a string.
GetSystemDefaultLangID	Retrieves the default language ID (LANGID) from a user's system. (1)
GetSystemDefaultLCID	Retrieves the default LCID from a user's system.
GetUserDefaultLangID	Retrieves the default LANGID from a user's system.
GetUserDefaultLCID	Retrieves the default LCID from a user's system. (1)

1 Because Windows is a single-user system, **GetUserDefaultLangID** and **GetUserDefaultLCID** return the same information as **GetSystemDefaultLangID** and **GetSystemDefaultLCID**.

Localized Member Names

An application may expose a set of objects whose members have names that differ across localized versions of a product. This poses a problem for programming languages that want to access such objects, because it means that late binding is sensitive to the locale of the application. The **IDispatch** and VTBL interfaces allow software developers a range of solutions that vary in cost of implementation and quality of national language support. All methods of the **IDispatch** interface that are potentially sensitive to language are passed an LCID.

Following are some of the possible approaches a class implementation may take:

- Accept any LCID and use the same member names in all locales. This is acceptable if the interface will typically be accessed only by advanced users. For example, the member names for OLE interfaces will never be localized.
- Simply return an error (DISP_E_UNKNOWNLCID) if the caller's LCID doesn't match the localized version of the class. This would prevent users from being able to write late-bound code which runs on machines with different localized implementations of the class.
- Recognize the particular version's localized names, as well as one language that is recognized in all versions. For example, a French version might accept French and English names, where English is the language supported in all versions. This would constrain users to use English when writing code that runs in all countries,.
- Accept all LCIDs supported by all versions of the product. This means that the implementation of **GetIDsOfNames** would need to interpret the passed array of names based on the given LCID. This is the preferred solution because users would be able to write code in their national language and run the code on any localized version of the application.

At the very least, the application must check the LCID before interpreting member names. Also note that the meaning of parameters passed to a member function may depend on the caller's national language. For example, a spreadsheet application might interpret the arguments to a **SetFormula** method differently, depending on the LCID.

Locale ID (LCID)

The **IDispatch** interface uses the 32-bit Windows definition of a LCID to identify locales. An LCID is a DWORD value that contains the LANGID in the lower word and a reserved value in the upper word. The bits are as follows:

{ewc msdncd, EWGraphic, bsd23531 0 /a "SDK.WMF"}

This LCID has the components necessary to uniquely identify one of the installed system-defined locales.

```
/*
 * LCID creation/extraction macros:
 * MAKELCID - construct locale ID from language ID and country code.
 */
#define MAKELCID(l) ((DWORD)(((WORD)(l))|(((DWORD)((WORD)(0))) << 16)))
```

There are two predefined LCID values. LOCALE_SYSTEM_DEFAULT is the system default locale, and LOCALE_USER_DEFAULT is the current user''s locale. However, when querying the NLS APIs for information, it is more efficient to query once for the current locale with **GetSystemDefaultLCID** or **GetUserDefaultLCID**, rather than using these constants.

Language ID (LANGID)

A LANGID is a 16-bit value that is the combination of a primary and sublanguage ID. The bits are as follows:

{ewc msdncd, EWGraphic, bsd23531 1 /a "SDK.WMF"}

Macros are provided for constructing a LANGID and extracting the fields:

LANGID creation/extraction macros include:

- MAKELANGID - construct LANGID from primary LANGID and sublanguage ID.
- PRIMARYLANGID - extract primary language ID from a LANGID.
- SUBLANGID - extract sublanguage ID from a LANGID.
- LANGIDFROMLCID - get the LANGID from an LCID.

```
#define MAKELANGID(p, s)            ((((USHORT)(s)) << 10) | (USHORT)(p))
#define PRIMARYLANGID(lgid)        ((USHORT)(lgid) & 0x3ff)
#define SUBLANGID(lgid)            ((USHORT)(lgid) >> 10)
#define LANGIDFROMLCID(lcid)        ((WORD)(lcid))
```

The following three combinations of primary and sublanguage IDs have special meanings:

PRIMARYLANGID	SUBLANGID	Meaning
LANG_NEUTRAL	SUBLANG_NEUTRAL	Language neutral
LANG_NEUTRAL	SUBLANG_SYS_DEFAULT	System default language
LANG_NEUTRAL	SUBLANG_DEFAULT	User default language

For primary language IDs, the range 0x200 to 0x3ff is user definable. The range 0x000 to 0x1ff is reserved for system use. The following table lists the primary language IDs supported by Automation:

Language	PRIMARYLANGID
Neutral	0x00
Chinese	0x04
Czech	0x05
Danish	0x06
Dutch	0x13
English	0x09
Finnish	0x0b
French	0x0c
German	0x07
Greek	0x08
Hungarian	0x0e
Icelandic	0x0F
Italian	0x10
Japanese	0x11
Korean	0x12
Norwegian	0x14

Polish	0x15
Portuguese	0x16
Russian	0x19
Serbo Croatian	0x1a
Slovak	0x1b
Spanish	0x0a
Swedish	0x1d
Turkish	0x1F

For sublanguage IDs, the range 0x20 to 0x3f is user definable. The range 0x00 to 0x1f is reserved for system use. The following table lists the sublanguage IDs supported by Automation:

Sublanguage	**SUBLANGID**
Neutral	0x00
Default	0x01
System Default	0x02
Chinese (Simplified)	0x02
Chinese (Traditional)	0x01
Dutch	0x01
Dutch (Belgian)	0x02
English (U.S.)	0x01
English (U.K.)	0x02
English (Australian)	0x03
English (Canadian)	0x04
English (Irish)	0x06
English (New Zealand)	0x05
French	0x01
French (Belgian)	0x02
French (Canadian)	0x03
French (Swiss)	0x04
German	0x01
German (Swiss)	0x02
German (Austrian)	0x03
Greek	0x01
Icelandic	0x01
Italian	0x01
Italian (Swiss)	0x02
Japanese	0x01
Korean	0x01
Norwegian (Bokmal)	0x01
Norwegian (Nynorsk)	0x02
Portuguese	0x02
Portuguese (Brazilian)	0x01
Serbo Croatian (Latin)	0x01
Spanish (Castilian)1	0x01
Spanish (Mexican)	0x02

Spanish (Modern)1	0x03
Turkish	0x01

1 The only difference between Spanish (Castilian) and Spanish (Modern) is the sort ordering. All of the LCType values are the same.

Locale Constants (LCTYPE)

An LCTYPE is a constant that specifies a particular piece of locale information. For example:

```
typedef  DWORD  LCTYPE;
```

The list of supported LCTYPES follows. All values are null-terminated, variable-length strings. Numeric values are expressed as strings of decimal digits, unless otherwise noted. The values in the brackets indicate the maximum number of characters allowed for the string (including the null termination). If no maximum is indicated, the string may be of variable length.

Constant name	Description
LOCALE_ILANGUAGE	A LANGID represented in hexadecimal digits. See the previous sections. [5]
LOCALE_SLANGUAGE	The full localized name of the language.
LOCALE_SENGLANGUAGE	The full English U.S. name of the language from the ISO Standard 639. This will always be restricted to characters that can be mapped into the ASCII 127-character subset.
LOCALE_SABBREVLANGNAME	The abbreviated name of the language, created by taking the two-letter language abbreviation, as found in ISO Standard 639, and adding a third letter as appropriate to indicate the sublanguage.
LOCALE_SNATIVELANGNAME	The native name of the language.
LOCALE_ICOUNTRY	The country code, based on international phone codes, also referred to as IBM country codes. [6]
LOCALE_SCOUNTRY	The full localized name of the country.
LOCALE_SENGCOUNTRY	The full English U.S. name of the country. This will always be restricted to characters that can be mapped into the ASCII 127-character subset.
LOCALE_SABBREVCTRYNAME	The abbreviated name of the country as found in ISO Standard 3166.
LOCALE_SNATIVECTRYNAME	The native name of the country.
LOCALE_IDEFAULTLANGUAGE	LANGID for the principal language spoken in this locale. This is provided so that partially specified locales can be completed with default values. [5]
LOCALE_IDEFAULTCOUNTRY	Country code for the principal

	country in this locale. This is provided so that partially specified locales can be completed with default values. [6]
LOCALE_IDEFAULTANSICODEPAGE	The ANSI code page associated with this locale. Format: 4 Unicode decimal digits plus a Unicode null terminator. [10] [6]
LOCALE_IDEFAULTCODEPAGE	The OEM code page associated with the country. [6]
LOCALE_SLIST	Characters used to separate list items. For example, a comma is used in many locales.
LOCALE_IMEASURE	This value is 0 for the metric system (S.I.) and 1 for the U.S. system of measurements. [2]
LOCALE_SDECIMAL	Characters used for the decimal separator.
LOCALE_STHOUSAND	Characters used as the separator between groups of digits left of the decimal.
LOCALE_SGROUPING	Sizes for each group of digits to the left of the decimal. An explicit size is required for each group. Sizes are separated by semicolons. If the last value is 0, the preceding value is repeated. To group thousands, specify **3;0**.
LOCALE_IDIGITS	The number of fractional digits. [3]
LOCALE_ILZERO	Whether to use leading zeros in decimal fields. [2] A setting of 0 means use no leading zeros; 1 means use leading zeros.
LOCALE_SNATIVEDIGITS	The ten characters that are the native equivalent of the ASCII 0-9.
LOCALE_INEGNUMBER	Negative number mode. [2] "0" (1.1) "1" -1.1 "2" -1.1 "3" 1.1 "4" 1.1
LOCALE_SCURRENCY	The string used as the local monetary symbol.
LOCALE_SINTLSYMBOL	Three characters of the International monetary symbol specified in ISO 4217, *Codes for the Representation of Currencies and Funds,* followed by the character separating this string

	from the amount.
LOCALE_SMONDECIMALSEP	Characters used for the monetary decimal separators.
LOCALE_SMONTHOUSANDSEP	Characters used as monetary separator between groups of digits left of the decimal.
LOCALE_SMONGROUPING	Sizes for each group of monetary digits to the left of the decimal. An explicit size is needed for each group. Sizes are separated by semicolons. If the last value is 0, the preceding value is repeated. To group thousands, specify **3;0**.
LOCALE_ICURRDIGITS	Number of fractional digits for the local monetary format. [3]
LOCALE_IINTLCURRDIGITS	Number of fractional digits for the international monetary format. [3]
LOCALE_ICURRENCY	Positive currency mode. [2] 0 Prefix, no separation. 1 Suffix, no separation. 2 Prefix, 1-character separation. 3 Suffix, 1-character separation.
LOCALE_INEGCURR	Negative currency mode. [2] 0 ($1.1) 1 -$1.1 2 $-1.1 3 $1.1- 4 $(1.1$) 5 -1.1$ 6 1.1-$ 7 1.1$- 8 -1.1 $ (space before $) 9 -$ 1.1 (space after $) 10 1.1 $- (space before $)
LOCALE_ICALENDARTYPE	The type of calendar currently in use. [2] 1 Gregorian (as in U.S.) 2 Gregorian (always English strings) 3 Era: Year of the Emperor (Japan) 4 Era: Year of the Republic of China 5 Tangun Era (Korea)
LOCALE_IOPTIONALCALENDAR	The additional calendar types available for this LCID. Can be a null-separated list of all valid optional calendars. [2] 0 None available

	1 Gregorian (as in U.S.) 2 Gregorian (always English strings) 3 Era: Year of the Emperor (Japan) 4 Era: Year of the Republic of China 5 Tangun Era (Korea)
LOCALE_SDATE	Characters used for the date separator.
LOCALE_STIME	Characters used for the time separator.
LOCALE_STIMEFORMAT	Time-formatting string. [80]
LOCALE_SSHORTDATE	Short Date_Time formatting strings for this locale.
LOCALE_SLONGDATE	Long Date_Time formatting strings for this locale.
LOCALE_IDATE	Short Date format-ordering specifier. [2] 0 Month - Day - Year 1 Day - Month - Year 2 Year - Month - Day
LOCALE_ILDATE	Long Date format ordering specifier. [2] 0 Month - Day - Year 1 Day - Month - Year 2 Year - Month - Day
LOCALE_ITIME	Time format specifier. [2] 0 AM/PM 12-hour format. 1 24-hour format.
LOCALE_ITIMEMARKPOSN	Whether the time marker string (AM|PM) precedes or follows the time string. (The registry value is named **ITimePrefix** for previous Far East version compatibility.) 0 Suffix (9:15 AM). 1 Prefix (AM 9:15).
LOCALE_ICENTURY	Whether to use full 4-digit century. [2] 0 Two digit. 1 Full century.
LOCALE_ITLZERO	Whether to use leading zeros in time fields. [2] 0 No leading zeros. 1 Leading zeros for hours.
LOCALE_IDAYLZERO	Whether to use leading zeros in day fields. [2] 0 No leading zeros. 1 Leading zeros.
LOCALE_IMONLZERO	Whether to use leading zeros in

	month fields. [2] 0 No leading zeros. 1 Leading zeros.
LOCALE_S1159	String for the AM designator.
LOCALE_S2359	String for the PM designator.
LOCALE_IFIRSTWEEKOFYEAR	Specifies which week of the year is considered first. [2] 0 Week containing 1/1 is the first week of the year. 1 First full week following 1/1is the first week of the year. 2 First week with at least 4 days is the first week of the year.
LOCALE_IFIRSTDAYOFWEEK	Specifies the day considered first in the week. [2] 0 SDAYNAME1 1 SDAYNAME2 2 SDAYNAME3 3 SDAYNAME4 4 SDAYNAME5 5 SDAYNAME6 6 DAYNAME7
LOCALE_SDAYNAME1	Long name for Monday.
LOCALE_SDAYNAME2	Long name for Tuesday.
LOCALE_SDAYNAME2	Long name for Tuesday.
LOCALE_SDAYNAME3	Long name for Wednesday.
LOCALE_SDAYNAME4	Long name for Thursday.
LOCALE_SDAYNAME5	Long name for Friday.
LOCALE_SDAYNAME6	Long name for Saturday.
LOCALE_SDAYNAME7	Long name for Sunday.
LOCALE_SABBREVDAYNAME1	Abbreviated name for Monday.
LOCALE_SABBREVDAYNAME2	Abbreviated name for Tuesday.
LOCALE_SABBREVDAYNAME3	Abbreviated name for Wednesday.
LOCALE_SABBREVDAYNAME4	Abbreviated name for Thursday.
LOCALE_SABBREVDAYNAME5	Abbreviated name for Friday.
LOCALE_SABBREVDAYNAME6	Abbreviated name for Saturday.
LOCALE_SABBREVDAYNAME7	Abbreviated name for Sunday.
LOCALE_SMONTHNAME1	Long name for January.
LOCALE_SMONTHNAME2	Long name for February.
LOCALE_SMONTHNAME3	Long name for March.
LOCALE_SMONTHNAME4	Long name for April.
LOCALE_SMONTHNAME5	Long name for May.
LOCALE_SMONTHNAME6	Long name for June.
LOCALE_SMONTHNAME7	Long name for July.
LOCALE_SMONTHNAME8	Long name for August.

LOCALE_SMONTHNAME9	Long name for September.
LOCALE_SMONTHNAME10	Long name for October.
LOCALE_SMONTHNAME11	Long name for November.
LOCALE_SMONTHNAME12	Long name for December.
LOCALE_SMONTHNAME13	Native name for 13th month, if it exists.
LOCALE_SABBREVMONTHNAME1	Abbreviated name for January.
LOCALE_SABBREVMONTHNAME2	Abbreviated name for February.
LOCALE_SABBREVMONTHNAME3	Abbreviated name for March.
LOCALE_SABBREVMONTHNAME4	Abbreviated name for April.
LOCALE_SABBREVMONTHNAME5	Abbreviated name for May.
LOCALE_SABBREVMONTHNAME6	Abbreviated name for June.
LOCALE_SABBREVMONTHNAME7	Abbreviated name for July.
LOCALE_SABBREVMONTHNAME8	Abbreviated name for August.
LOCALE_SABBREVMONTHNAME9	Abbreviated name for September.
LOCALE_SABBREVMONTHNAME10	Abbreviated name for October.
LOCALE_SABBREVMONTHNAME11	Abbreviated name for November.
LOCALE_SABBREVMONTHNAME12	Abbreviated name for December.
LOCALE_SABBREVMONTHNAME13	Native abbreviated name for 13th month, if it exists.
LOCALE_SPOSITIVESIGN	String value for the positive sign.
LOCALE_SNEGATIVESIGN	String value for the negative sign.
LOCALE_IPOSSIGNPOSN	Formatting index for positive values. [2] 0 Parentheses surround the amount and the monetary symbol. 1 The sign string precedes the amount and the monetary symbol. 2 The sign string precedes the amount and the monetary symbol. 3 The sign string precedes the amount and the monetary symbol. 4 The sign string precedes the amount and the

	monetary symbol.
LOCALE_INEGSIGNPOSN	Formatting index for negative values. [2] 0 Parentheses surround the amount and the monetary symbol. 1 The sign string precedes the amount and the monetary symbol. 2 The sign string precedes the amount and the monetary symbol. 3 The sign string precedes the amount and the monetary symbol. 4 The sign string precedes the amount and the monetary symbol.
LOCALE_IPOSSYMPRECEDES	If the monetary symbol precedes, 1. If it succeeds a positive amount, 0. [2]
LOCALE_IPOSSEPBYSPACE	If the monetary symbol is separated by a space from a positive amount, 1. Otherwise, 0. [2]
LOCALE_INEGSYMPRECEDES	If the monetary symbol precedes, 1. If it succeeds a negative amount, 0. [2]
LOCALE_INEGSEPBYSPACE	If the monetary symbol is separated by a space from a negative amount, 1. Otherwise, 0. [2]

The following table shows the equivalence between LCTYPE values and the information stored in the [intl] section of Win.ini. These values are retrieved from Win.ini if information for the current system locale is queried. Values for LCTYPEs that are not in the following table do not depend on information stored in Win.ini.

Win.ini settings	**LCTYPE**
sLanguage (1)	LOCALE_SABBREVLANGNAME
iCountry	LOCALE_ICOUNTRY
sCountry	LOCALE_SCOUNTRY
sList	LOCALE_SLIST
iMeasure	LOCALE_IMEASURE
sDecimal	LOCALE_SDECIMAL
sThousand	LOCALE_STHOUSAND
iDigits	LOCALE_IDIGITS
iLZero	LOCALE_ILZERO
sCurrency	LOCALE_SCURRENCY
iCurrDigits	LOCALE_ICURRDIGITS

iCurrency	LOCALE_ICURRENCY
iNegCurr	LOCALE_INEGCURR
sDate	LOCALE_SDATE
sTime	LOCALE_STIME
sShortDate	LOCALE_SSHORTDATE
sLongDate	LOCALE_SLONGDATE
iDate	LOCALE_IDATE
iTime	LOCALE_ITIME
iTLZero	LOCALE_ITLZERO
s1159	LOCALE_S1159
s2359	LOCALE_S2359

1 Unlike in Win.ini, values returned by LOCALE_SABBREVLANGNAME are always in uppercase.

CompareStringA

int CompareStringA(

LCID lcid
DWORD dwCmpFlags
LPCSTR lpString1
integer cchCount1
LPCSTR lpString2
integer cchCount2

Compares two character strings of the same locale according to the supplied LCID.

Parameters

lcid

Locale context for the comparison. The strings are assumed to be represented in the default ANSI code page for this locale.

dwCmpFlags

Flags that indicate the character traits to use or ignore when comparing the two strings. Several flags can be combined , or none can be used. (In the case of this function, there are no illegal combinations of flags.) Compare flags include the following.

Value	Meaning
NORM_IGNORECASE	Ignore case. Default is Off.
NORM_IGNOREKANATYPE	Ignore Japanese hiragana/katakana character differences. Default is Off.
NORM_IGNORENONSPACE	Ignore nonspacing marks (accents, diacritics, and vowel marks). Default is Off.
NORM_IGNORESYMBOLS	Ignore symbols. Default is Off.
NORM_IGNOREWIDTH	Ignore character width. Default is Off.

lpString1 and *lpString2*

The two strings to be compared.

cchCount1 and *cchCount2*

The character counts of the two strings. The count does not include the null-terminator (if any). If either *cchCount1* or *cchCount2* is -1, the corresponding string is assumed to be null-terminated, and the length is calculated automatically.

Return Value

Value	Meaning
0	Failure.
1	*lpString1* is less than *lpString2*.
2	*lpString1* is equal to *lpString2*.

3	*lpString1 is greater than lpString2*.

Comments

When used without any flags, this function uses the same sorting algorithm as *lstrcmp* in the given locale. When used with NORM_IGNORECASE, the same algorithm as *lstrcmpi* is used.

For double-byte character set (DBCS) locales, the flag NORM_IGNORECASE has an effect on all the wide (two-byte) characters as well as the narrow (one-byte) characters. This includes the wide Greek and Cyrillic characters.

In Chinese Simplified, the sorting order used to compare the strings is based on the following sequence: symbols, digit numbers, English letters, and Chinese Simplified characters. The characters within each group sort in character-code order.

In Chinese Traditional, the sorting order used to compare the strings is based on the number of strokes in the characters. Symbols, digit numbers, and English characters are considered to have zero strokes. The sort sequence is symbols, digit numbers, English letters, and Chinese Traditional characters. The characters within each stroke-number group sort in character-code order.

In Japanese, the sorting order used to compare the strings is based on the Japanese 50-on sorting sequence. The Kanji ideographic characters sort in character-code order.

In Japanese, the flag NORM_IGNORENONSPACE has an effect on the daku-on, handaku-on, chou-on, you-on, and soku-on modifiers, and on the repeat kana/kanji characters.

In Korean, the sort order is based on the sequence: symbols, digit numbers, Jaso and Hangeul, Hanja, and English. Within the Jaso-Hangeul group, each Jaso character is followed by the Hangeuls that start with that Jaso. Hanja characters are sorted in Hangeul pronunciation order. Where multiple Hanja have the same Hangeul pronunciation, they are sorted in character-code order.

The NORM_IGNORENONSPACE flag only has an effect for the locales in which accented characters are sorted in a second pass from main characters. All characters in the string are first compared without regard to accents and (if the strings are equal) a second pass over the strings to compare accents is performed. In this case, this flag causes the second pass to not be performed. Some locales sort accented characters in the first pass, in which case this flag will have no effect.

If the return value is 2, the two strings are equal in the collation sense, though not necessarily identical (the case might be ignored, and so on).

If the two strings are of different lengths, they are compared up to the length of the shortest one. If they are equal to that point, the return value will indicate that the longer string is greater.

To maintain the C run-time convention of comparing strings, the value 2 can be subtracted from a non-zero return value. The meaning of < 0, == 0, and > 0 is then consistent with the C run-time conventions.

LCMapStringA

int LCMapStringA(

LCID lcid
DWORD dwMapFlags
LPCSTR lpSrcStr
int cchSrc
LPSTR lpDestStr
int cchDest

Transforms the case or sort order of a string.

Parameters

lcid

Locale ID context for mapping. The strings are assumed to be represented in the default ANSI code page for this locale.

dwMapFlags

Flags that indicate what type of transformation is to occur during mapping. Several flags can be combined on a single transformation (though some combinations are illegal). Mapping options include the following.

Name	Meaning
LCMAP_LOWERCASE	Lowercase.
LCMAP_UPPERCASE	Uppercase.
LCMAP_SORTKEY	Character sort key.
LCMAP_HALFWIDTH	Narrow characters (where applicable).
LCMAP_FULLWIDTH	Wide characters (where applicable).
LCMAP_HIRAGANA	Hiragana.
LCMAP_KATAKANA	Katakana.
NORM_IGNORECASE	Ignore case. Default is Off.
NORM_IGNORENONSPACE	Ignore nonspacing. Default is Off.
NORM_IGNOREWIDTH	Ignore character width. Default is Off.
NORM_IGNOREKANATYPE	Ignore Japanese hiragana/katakana character differences. Default is Off.
NORM_IGNORESYMBOLS	Ignore symbols. Default is Off.

The latter five options (NORM_IGNORECASE, NORM_IGNORENONSPACE, NORM_IGNOREWIDTH, NORM_IGNOREKANATYPE, and NORM_IGNORESYMBOLS) are normalization options that can only be used in combination with the LCMAP_SORTKEY conversion option.

Conversion options can be combined only when they are taken from the following three groups, and then only when there is no more than one option from each group:

- Casing options (LCMAP_LOWERCASE, LCMAP_UPPERCASE)
- Width options (LCMAP_HALFWIDTH, LCMAP_FULLWIDTH)
- Kana options (LCMAP_HIRAGANA, LCMAP_KATAKANA)

lpSrcStr

Pointer to the supplied string to be mapped.

cchSrc

Character count of the input string buffer. If -1, *lpSrcStr* is assumed to be null-terminated and the length is calculated automatically.

lpDestStr

Pointer to the memory buffer that stores the resulting mapped string.

cchDest

Character count of the memory buffer pointed to by *lpDestStr*. If *cchDest* is 0, then the return value of this function is the number of characters required to hold the mapped string. In this case, the *lpDestStr* pointer is not referenced.

Return Value

Value	Meaning
0	Failure.
The number of characters written to *lpDestSt*	Success.

Comments

LCMapStringA maps one character string to another, performing the specified locale-dependent translation.

The flag LCMAP_UPPER produces the same result as **AnsiUpper** in the given locale. The flag LCMAP_LOWER produces the same result as **AnsiLower.** This function always maps a single character to a single character.

The mapped string is null-terminated if the source string is null-terminated.

When used with LCMAP_UPPER and LCMAP_LOWER, the *lpSrcStr* and *lpDestStr* may be the same to produce an in-place mapping. When LCMAP_SORTKEY is used, the *lpSrcStr* and *lpDestStr* pointers may not be the same. In this case, an error will result.

The LCMAP_SORTKEY transforms two strings so that when they are compared with the standard C library function **strcmp** (by strict numerical valuation of their characters), the same order will result, as if the original strings were compared with **CompareStringA**. When LCMAP_SORTKEY is specified, the output string is a string (without Nulls, except for the terminator), but the character values will not be meaningful display values. This is similar behavior to the ANSI C function **strxfrm**.

GetLocaleInfoA

int GetLocaleInfoA(

LCID *lcid*
LCTYPE *LCType*
LPSTR *lpLCData*
int *cchData*

Retrieves locale information from the user's system.

Parameters

lcid

The locale ID. The returned string is represented in the default ANSI code page for this locale.

LCType

Flag that indicates the type of information to be returned by the call. See the listing of constant values defined in this chapter. LOCALE_NOUSEROVERRIDE **|** LCTYPE indicates that the desired information will always be retrieved from the locale database, even if the LCID is the current one and the user has changed some of the values in the Windows 95 Control Panel. If this flag is not specified, the values in Win.ini take precedence over the database settings when getting values for the current system default locale.

lpLCData

Pointer to the memory where **GetLocaleInfoA** will return the requested data. This pointer is not referenced if *cchData* is 0.

cchData

Character count of the supplied *lpLCData* memory buffer. If *cchData* is 0, the return value is the number of characters required to hold the string, including the terminating null character. In this case, *lpLCData* is not referenced.

Return Value

Value	Meaning
0	Failure.
The number of characters copied, including the terminating null character	Success.

Comments

GetLocaleInfoA returns one of the various pieces of information about a locale by querying the stored locale database or Win.ini. The call also indicates how much memory is necessary to contain the desired information.

The information returned is always a null-terminated string. No integers are returned by this function and numeric values are returned as text. (See the format descriptions under LCTYPE).

GetStringTypeA Quick Info

BOOL GetStringTypeA(

LCID lcid
DWORD dwInfoType
LPCSTR lpSrcStr
int cchSrc
LPWORD lpCharType

Retrieves locale type information about each character in a string.

Parameters

lcid

Locale context for the mapping. The string is assumed to be represented in the default ANSI code page for this locale.

dwInfoType

Type of character information to retrieve. The various types are divided into different levels. (See the Comments section for a list of information included in each type). The options are mutually exclusive. The following types are supported:

- CT_CTYPE1
- CT_CTYPE2
- CT_CTYPE3

lpSrcStr

String for which character types are requested. If *cchSrc* is -1, *lpSrcStr* is assumed to be null-terminated.

cchSrc

Character count of *lpSrcStr*. If *cchSrc* is -1, *lpSrcStr* is assumed to be null-terminated. This must also be the character count of *lpCharType*.

lpCharType

Array of the same length as *lpSrcStr* (*cchSrc*). On output, the array contains one word corresponding to each character in *lpSrcStr*.

Return Value

Return value	Meaning
0	Failure.
1	Success.

Comments

The *lpSrcStr* and *lpCharType* pointers cannot be the same. In this case, the error ERROR_INVALID_PARAMETER results.

The character type bits are divided up into several levels. One level's information can be retrieved by a single call.

This function supports three character types:

- Ctype 1
- Ctype 2
- Ctype 3

Ctype 1 character types support ANSI C and POSIX character typing functions. A bitwise OR of these values is returned when *dwInfoType* is set to CT_CTYPE1. For DBCS locales, the Ctype 1 attributes apply to both narrow characters and wide characters. The Japanese hiragana and katakana characters, and the kanji ideograph characters all have the C1_ALPHA attribute.

The following table lists the Ctype 1 character types.

Name	Value	Meaning
C1_UPPER	0x0001	Uppercase1.
C1_LOWER	0x0002	Lowercase1.
C1_DIGIT	0x0004	Decimal digits.
C1_SPACE	0x0008	Space characters.
C1_PUNCT	0x0010	Punctuation.
C1_CNTRL	0x0020	Control characters.
C1_BLANK	0x0040	Blank characters.
C1_XDIGIT	0x0080	Hexadecimal digits.
C1_ALPHA	0x0100	Any letter.

1 The Windows version 3.1 functions **IsCharUpper** and **IsCharLower** do not always produce correct results for characters in the range 0x80-0x9f, so they may produce different results than this function for characters in that range. (For example, the German Windows version 3.1 language driver incorrectly reports 0x9a, lowercase s hacek, as uppercase).

Ctype 2 character types support the proper layout of text. For DBCS locales, Ctype 2 applies to both narrow and wide characters. The directional attributes are assigned so that the BiDi layout algorithm standardized by Unicode produces the correct results. For more information on the use of these attributes, see *The Unicode Standard: Worldwide Character Encoding* from Addison-Wesley publishers.

	Name	Value	Meaning
Strong	C2_LEFTTORIGHT	0x1	Left to right.
	C2_RIGHTTOLEFT	0x2	Right to left.
Weak	C2_EUROPENUMBER	0x3	European number, European digit.
	C2_EUROPESEPARATOR	0x4	European numeric separator.
	C2_EUROPETERMINATOR	0x5	European numeric terminator.
	C2_ARABICNUMBER	0x6	Arabic number.
	C2_COMMONSEPARATOR	0x7	Common numeric separator.
Neutral	C2_BLOCKSEPARATOR	0x8	Block separator.

	C2_SEGMENTSEPARATOR	0x9	Segment separator.
	C2_WHITESPACE	0xA	White space.
	C2_OTHERNEUTRAL	0xB	Other neutrals.
Not applicable	C2_NOTAPPLICABLE	0x0	No implicit direction (for example, control codes).

Ctype 3 character types are general text-processing information. A bitwise OR of these values is returned when *dwInfoType* is set to CT_CTYPE3. For DBCS locales, the Ctype 3 attributes apply to both narrow characters and wide characters. The Japanese hiragana and katakana characters, and the kanji ideograph characters all have the C3_ALPHA attribute.

Name	**Value**	**Meaning**
C3_NONSPACING	0x1	Nonspacing mark.
C3_DIACRITIC	0x2	Diacritic nonspacing mark.
C3_VOWELMARK	0x4	Vowel nonspacing mark.
C3_SYMBOL	0x8	Symbol.
C3_KATAKANA	0x10	Katakana character.
C3_HIRAGANA	0x20	Hiragana character.
C3_HALFWIDTH	0x40	Narrow character.
C3_FULLWIDTH	0x80	Wide character.
C3_IDEOGRAPH	0x100	Ideograph.
C3_ALPHA	0x8000	Any letter.
C3_NOTAPPLICABLE	0x0	Not applicable.

GetSystemDefaultLangID Quick Info

LANGID GetSystemDefaultLangID(

Retrieves the default LANGID from a user's system.

Return Value

Return value	Meaning
0	Failure.
System default LANGID	Success.

Comments

Returns the system default LANGID. For information on how this value is determined, see **GetSystemDefaultLCID** in the following section..

GetSystemDefaultLCID Quick Info

LCID GetSystemDefaultLCID(

Retrieves the default LCID from a user's system.

Return Value

Return value	Meaning
0	Failure.
System default locale ID	Success.

Comments

Returns the system default LCID. The return value is determined by examining the values of *sLanguage* and *iCountry* in Win.ini, and comparing the values to those in the stored locale database. If no matching values are found, or the required values cannot be read from Win.ini, or if the stored locale database cannot be loaded, the value 0 is returned.

GetUserDefaultLangID Quick Info

LANGID GetUserDefaultLangID(

Retrieves the default LANGID from a user's system.

Return Value

Value	Meaning
0	Failure.
User default LANGID	Success.

Comments

Returns the user default LANGID. On single-user systems, the value returned from this function is always the same as that returned from **GetSystemDefaultLangID**.

GetUserDefaultLCID Quick Info

LCID GetUserDefaultLCID(

Retrieves the default LCID from a user's system.

Return Value

Return value	Meaning
0	Failure.
User default locale ID	Success.

Comments

Returns the user default LCID. On single-user systems, the value returned by this function is always the same as that returned from **GetSystemDefaultLCID**.

File Requirements

The file names shown in **bold** are required by your application at run time.

32-bit filenames	16-bit filenames	Purpose
None	Ole2.reg	Registers OLE and Automation. OLE is a system component on 32-bit systems, therefore, no .reg file is required.
None	**Ole2nls.dll** Ole2nls.lib Olenls.h	Provides functions for applications that support multiple national languages. On 32-bit systems, NLS features are provided by the Win32 NLS API.
Oleprx32.dll	**Ole2prox.dll**	Coordinates object access across processes.
MkTypLib.exe	MkTypLib.exe	Builds type libraries from interface descriptions.
Oleaut32.dll Oleaut32.lib Oleauto.h	**TypeLib.dll** Dispatch.h	Accesses type libraries.
	Ole2disp.dll Ole2disp.lib Dispatch.h	Provides functions for creating ActiveX objects and retrieving active objects at run time. Accesses ActiveX objects by invoking methods and properties.
Ole32.dll Ole32.lib Ole2.h	**Ole2.dll** Ole2.lib Ole2.h	Provides OLE functions that can be used by OLE and ActiveX objects or containers.
	Compobj.dll Compobj.lib Ole2.h Compobj.h	Supports COM creation and access.
	Storage.dll Storage.lib Ole2.h Storage.h	Supports access to subfiles, such as type libraries, within compound documents.

Information for Visual Basic Programmers

Visual Basic provides full support for Automation. The following table lists how Visual Basic statements translate into OLE APIs.

Visual Basic statement	OLE APIs
CreateObject	**CLSIDFromProgID**
	CoCreateInstance
	QueryInterface to get **IDispatch** interface.
GetObject	**CLSIDFromProgID**
	CoCreateInstance
	QueryInterface for **IPersistFile** interface.
	Load on **IPersistFile** interface.
	QueryInterface to get **IDispatch** interface.
GetObject	**CreateBindCtx** creates the bind context for the subsequent functions.
	MkParseDisplayName returns a moniker handle for **BindMoniker**.
	BindMoniker returns a pointer to the **IDispatch** interface.
	Release on moniker handle.
	Release on context.
GetObject	**CLSIDFromProgID**
	GetActiveObject on class ID.
	QueryInterface to get **IDispatch** interface.
Dim *x* **As New** *interface*	Find CLSID for *interface*.
	CoCreateInstance
	QueryInterface

String Comparisons

This appendix describes how Automation compares strings. These comparisons are important when creating applications that support national language accents and digraphs. The information in this appendix applies to the following:

- **CreateStdDispatch**
- **DispGetIDsOfNames**
- **ITypeLib::FindName**
- **ITypeLib::GetIDsOfNames**
- **MkTypLib and the MIDL compiler**

When comparing strings, Automation components use the following rules:

- Comparisons are sensitive to locale, based on the string's locale ID (LCID). A string must have an LCID that is supported by the application or type library. For more information about locales and LCIDs, refer to the section "Supporting Multiple National Languages," in Chapter 2, "Exposing ActiveX Objects."
- Accent characters are ignored. For example, the string **à** compares the same as **a**.
- Case is ignored. For example, the string **A** compares the same as **a**.
- Comparisons are sensitive to digraphs. For example, the string **Æ** is not the same as **AE**.
- For Japanese, Korean, and Chinese locales, **ITypeLib::FindName** and **ITypeLib::GetIDsOfNames** ignore width and kanatype.

Managing GUIDs

Globally unique identifiers (GUIDs) appear in many places in a typical Automation application. GUID errors can cause persistent bugs. To help avoid GUID problems, this appendix lists all of the places that GUIDs appear in a typical Automation application, describes common characteristics of GUID bugs, and offers some GUID management techniques.

GUIDs are the same as UUIDs (Universally Unique Identifiers). A class identifier (CLSID) is a UUID/GUID that refers to a class.

The System Registry

The system registry is a central repository that contains information about objects. GUIDs are used to index that information. You can view the registration information on your system by running Regedit.exe with the **/v** option:

```
regedit /v
```

Typically, a name is connected with a GUID (for example, Hello.Application maps to a GUID), and the GUID is connected to all the other relevant aspects of the object (for example, the GUID maps to Hello.exe).

GUIDs are created with the tool Guidgen.exe. Running Guidgen.exe produces a very large hexidecimal number that uniquely identifies an object, whether it is a class, interface, library, or some other type of object.

GUID Locations

GUIDs appear in the following locations:

- **.reg files** – When an application is created, usually one or more .reg files are created. The .reg files contain the GUIDs for the classes that an application exposes. These GUIDs are added to the registry when you run Regedit.exe to register the classes, or when you register type information with **LoadTypeLib**.
- **The system registry** – Contains the GUIDs for classes in multiple locations. This is where OLE and applications get information about classes.
- **.odl files** – When objects are described in an Object Description Language (.odl) file, a GUID needs to be provided for each object. Compiling the .odl file with the MIDL compiler or the MkTypLib utility places the GUIDs in a type library, which usually exists as a file with a .tlb extension. If a GUID is changed in an .odl file, you should run MIDL or MkTypLib again.
- **.tlb files** – Type libraries describe classes, and this information includes the GUIDs for the classes. The .tlb files can be browsed using the Tlbrowse.exe sample application supplied with OLE.
- **.h files** – Most application developers will declare class IDs (CLSIDs) for their classes in a header file by using the DEFINE_GUID macro.

Troubleshooting

The following problems are common with GUIDs.

Problem

GetObject can't seem to create an instance of my application.

Solution

Visual Basic uses the OLE calls listed in Appendix C to find the .exe file that creates an application instance.

Quick Info **Visual Basic proceeds as follows**

1. Looks up the GUID for the object. (For the Hello application, Visual Basic maps the programmatic ID (ProgID) Hello.Application into a GUID.)
2. Finds the object's server (the Hello.exe for the Hello application).
3. Launches the application.

If an error occurs, check the following:

- Has the .reg file been run?
- Are the entries in the registry correct?
- Do all of the GUIDs match?
- Can the application be launched? The .exe file for the application, listed in the LocalServer entry, should either be on the path or it should be fully specified. For example:

```
c:\ole2\sample\hello\hello.bin
```

Problem

When I use **GetObject,** the application launches, but the **GetObject** call fails.

Solution

Normally, when an application is started, a class factory is registered using **CoRegisterClassObject**. Some applications register their class factories only when launched with the **/Automation** switch. If code was inherited, or a sample was copied, determine whether it checks for this switch. The **/Automation** option might appear in the .reg file, the registry, or in the development environment.

Problem

GetTypeInfoOfGuid() fails to get the type information from my type library.

Solution

When **GetTypeInfoOfGuid** is called, a GUID is provided. If this GUID doesn't match the GUID in the .tlb file, no type information will be returned. The GUID in the code may be declared in a header file. The GUID in the .tlb file can be checked by using Browse.exe, which is provided with OLE, or with the Visual Basic Object Browser.

GUID Management

The problem with managing GUIDs is that they are pervasive, and their length prohibits simple comparisons.

The single most important technique in managing GUIDs is to keep a central list of all the GUIDs that are implemented. For example, use the DEFINE_GUID macro or the Guidgen.exe tool with the **-n** option to generate the required number of GUIDs, and then enter the resulting strings in the first column of a spreadsheet. Each time a new GUID is used, enter a description of its purpose in the second column of the spreadsheet.

> **Note** The DEFINE_GUIDE macro does not generate GUIDs. It defines a 128-bit number to a GUID with a human-readable name.

A central spreadsheet of GUIDs has several advantages:

- Listing all the GUIDs in one location may prevent accidental reuse of a GUID. (This often happens when an application is cloned to create another one.)
- The spreadsheet can be used to compare GUIDs. To check the accuracy of a GUID, you can copy it from the location where it is being used (for example, a .reg file), paste it into the spreadsheet, and then compare the two cells with the **=** operator.
- A record of GUID usage can be helpful in case of future problems, and this single source of information will be available to find the GUID for an object.

A

accessor function
A function that sets or retrieves the value of a property. Most properties have a pair of accessor functions. Properties that are read-only may have only one accessor function.

ActiveX
Microsoft's brand name for the technologies that enable interoperability using the Component Object Model (COM). ActiveX technology includes, but is not limited to, OLE.

ActiveX client
Any program or piece of code that accesses the functionality and the content of an ActiveX or OLE object.

ActiveX component
Physical file that contains classes, which are definitions of objects. For example, a .dll, .exe or .ocx file.

ActiveX control
A user interface element created using ActiveX technology.

ActiveX object
Objects an application or programming tool exposes to ActiveX clients.

Application object
The top-level object in an application's object hierarchy. The Application object identifies the application to the system, and typically becomes active when the application starts. Specified by the **appobj** attribute in the type library.

Automation
COM-based technology that enables interoperability among ActiveX components, including OLE components. Formerly referred to as OLE Automation.

Automation controller
An application, programming tool, or scripting language that accesses Automation objects. Visual Basic is an OLE Automation controller.

Automation object
An instance of a class defined within an application that is exposed for access by other applications or programming tools by Automation interfaces.

Automation server
An application, type library, or other source that makes Automation objects available for programming by other applications, programming tools, or scripting languages.

C

class identifier (CLSID)

A universally unique ID (UUID) for an application class that identifies an object. An object registers its class ID (CLSID) in the system registration database so that it can be loaded and programmed by other applications.

class factory

An object that implements the **IClassFactory** interface, which allows it to create other objects of a specific class.

coclass

Component object model class. A top-level object in the object hierarchy.

collection object

A grouping of exposed objects. A collection object that can address multiple occurrences of an object as a unit (for example, to draw a set of points).

Component Object Model (COM)

The programming model and binary standard on which OLE is based. COM defines how objects and their clients interact within processes or across process boundaries.

compound document

A document that contains data of different formats, such as sound clips, spreadsheets, text, and bitmaps, created by different applications. Compound documents are stored by container applications.

container application

An OLE-based application that provides storage, a display site, and access to a compound document object.

D

Dispatch identifier (DISPID)
The number by which a member function, parameter, or data member of an object is known internally to the **IDispatch** interface.

dispinterface
An **IDispatch** interface that responds only to a certain fixed set of names. The properties and methods of the dispinterface are not in the virtual function table (VTBL) for the object.

dual interface
An interface that supports both **IDispatch** and VTBL binding.

E

event

An action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. In Automation, an event is a method that is called, rather than implemented, by an Automation object.

event sink

A function that handles events. The code associated with a Visual Basic form, which contains event handlers for one or more controls, is an event sink.

event source

A control that experiences events and calls an event handler to dispose of them.

exposed object

See Automation object.

H

HRESULT

A value returned from a function call to an interface, consisting of a severity code, context information, a facility code, and a status code that describes the result. For 16-bit Windows systems, the HRESULT is an opaque result handle defined to be zero for a successful return from a function, and nonzero if error or status information is to be returned. To convert an HRESULT into a more detailed SCODE (or return value), applications call **GetSCode()**. *See SCODE.*

I

ID binding

The ability to bind member names to dispatch IDs (DISPIDs) at compile time (for example, by obtaining the IDs from a type library). This approach eliminates the need for calls to **IDispatch::GetIDsOfNames,** and results in improved performance over late-bound calls. *See also late binding.*

in-place activation

The ability to activate an object from within an OLE control and to associate a verb with that activation (for example, edit, play, change). Sometimes referred to as in-place editing or visual editing.

in-process server

An object application that runs in the same process space as the Automation controller.

interface

One or more well-defined base classes providing member functions that, when implemented in an application, provide a specific service. Interfaces may include compiled support functions to simplify their implementation.

L

late binding

The ability to bind names to IDs at run time, rather than at compile time.

locale identifier (LCID)

A 32-bit value that identifies the human language preferred by a user, region, or application.

M

marshaling

The process of packaging and sending interface parameters across process boundaries.

member function

One of a group of related functions that make up an interface. *See also method and property.*

method

A member function of an exposed object that performs some action on the object, such as saving it to disk.

MIDL compiler

The Microsoft Interface Definition Library (MIDL) compiler can be used to generate a type library. For information about the MIDL compiler, refer to the *Microsoft Interface Definition Language Programmer's Guide and Reference* in the Win32 Software Development Kit (SDK) section of the Microsoft Developer's Network (MSDN).

multiple-document interface (MDI) application

An application that can support multiple documents from one application instance. MDI object applications can simultaneously service a user and one or more embedding containers. *See also single-document interface (SDI) application.*

O

object

A unit of information that resides in a compound document and whose behavior is constant no matter where it is located or used.

Object Description Language (ODL)

A scripting language used to describe exposed libraries, objects, types, and interfaces. ODL scripts are compiled into type libraries by the MkTypLib tool.

OLE

Microsoft's object-based technology for sharing information and services across process and machine boundaries (object linking and embedding).

Out-of process server

An object application implemented in an executable file that runs in a separate process space from the Automation controller.

P

programmable object
See Automation object.

property
A data member of an exposed object. Properties are set or returned by means of get and put accessor functions.

proxy
An interface-specific object that packages parameters for that interface in preparation for a remote method call. A proxy runs in the address space of the sender and communicates with a corresponding stub in the receiver's address space. *See also stub, marshaling, and unmarshaling.*

S

safe array

An array that contains information about the number of dimensions and the bounds of its dimensions. Safe arrays are passed by **IDispatch::Invoke** within VARIANTARGs. Their base type is VT_tag | VT_ARRAY.

SCODE

A DWORD value that is used in 16-bit systems to pass detailed information to the caller of an interface member or API function. The status codes for OLE interfaces and APIs are defined in FACILITY_ITF. *See HRESULT.*

single-document interface (SDI) application

An application that can support only one document at a time. Multiple instances of an SDI application must be started to service both an embedded object and a user. *See also multiple-document interface (MDI) application*.

stub

An interface-specific object that unpackages the parameters for that interface after they are marshaled across the process boundary, and makes the requested method call. The stub runs in the address space of the receiver and communicates with a corresponding proxy in the sender's address space. *See proxy, marshaling, and unmarshaling.*

type description

The information used to build the type information for one or more aspects of an application's interface. Type descriptions are written in Object Description Language (ODL), and include both programmable and nonprogrammable interfaces.

type information

Information that describes the interfaces of an application. Type information is created from type descriptions using OLE Automation tools, such as MkTypLib or the **CreateDispTypeInfo** function. Type information can be accessed through the **ITypeInfo** interface.

type information element

A unit of information identified by one of these statements in a type description: **typedef, enum, struct, module, interface, dispinterface,** or **coclass**.

type library

A file or component within another file that contains type information about exposed objects. Type libraries are created from type descriptions using MkTypLib, and can be accessed through the **ITypeLib** interface.

U

unmarshaling
The process of unpackaging parameters that have been sent across process boundaries.

V

Value property

The property that defines the default behavior of an object when no other methods or properties are specified. Indicate the Value property by specifying the **default** attribute in ODL.

virtual function table (VTBL)

A table of function pointers, such as an implementation of a class in C++. The pointers in the VTBL point to the members of the interfaces that an object supports.