Introduction

Besides Automation, there is an alternate way to use COM with the PowerBASIC compilers: Direct calls to the methods and properties implemented in an interface through his VTable (short for Virtual Table, an array of pointers holding the addresses of the functions). This technique also allows you to use interfaces that derive directly from the IUnknown interface and that can't be used with Automation because don't have a dispatch interface. These include all the standard COM interfaces. His only limitattion is that can't be used with dispatch-only interfaces.

A dual interface enables direct vtable access to all its functions while a dispatch interface does not. A PowerBASIC client can query for a dual interface pointer and use direct vtable access to invoke its functions. This provides faster access than invoking the function using the IDispatch::GetIDsOfNames/or and IDispatch::Invoke functions. This may be important or not depending of the time used by the called function to perform his task, that is, calling the function directly has much less overhead but the called function doesn't execute faster (sorry for the obviety).

The Component Object Model

The Microsoft Component Object Model (COM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents), ActiveX® (Internet-enabled components), as well as others.

To understand COM (and therefore all COM-based technologies), it is crucial to understand 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. Rather, COM specifies an object model and programming requirements that enable COM objects (also called COM components, or sometimes simply *objects*) to interact with other objects. These objects can be within a single process, in other processes, and can even be on remote machines. They can have been written in other languages, and they may be structurally quite dissimilar, which is why COM is referred to as a *binary standard*—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 Microsoft® Visual 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, and it provides a small number of API functions that all components require. COM also defines how objects work together over a distributed environment and has added security features to help provide system and component integrity.

COM Objects and Interfaces

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

COM uses the word *interface* in a sense different from that typically used in Visual 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 a specific interface does not necessarily represent all the functions that the class supports. (Java programmers will find themselves at home with COM interfaces because Java defines interfaces in just the same way.)

Referring to an object *implementing* 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.

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 can use in an object-based application are included in the COM libraries. However, programmers are 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 the implementation provides the services that the object offers.

For example, a hypothetical interface named IStack 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 would not specify how the functions are to be implemented in code. In implementing the interface, 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, the in-memory representation of a pointer to an IStack interface, and therefore its use by a client, is completely determined by the interface definition.

Simple objects support only a single interface. More complicated objects, such as embeddable objects, typically support several interfaces. Clients have access to a COM 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 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 that interface's 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.

Interface Pointers and Interfaces

An instance of an interface implementation is actually a pointer to an array of pointers to methods—that is, 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 by using Visual 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, the compiler, given the names of methods, can check the types of parameters and return values of each interface method call. In contrast, if a client uses a position-based calling scheme, such type-checking is not available, even in C or C++.

Each interface—the immutable contract of a functional group of methods—is referred to at run time with a globally unique interface identifier (IID). This IID, which is a specific instance of a globally unique identifier (GUID) supported by COM, allows a client to ask an object precisely whether 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 a COM interface is, and is not:

- A COM interface is not the same as a C++ class—The pure virtual definition carries no implementation. If you are a C++ programmer, you can 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.

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

- COM 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.

- COM interfaces are immutable—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. However, objects can 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, and systemwide 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.

The IUnknown interface

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.

The following wrapper functions allow to call the three methods of the IUnknown interface with the PowerBASIC compilers. The *pthis* parameter will be address of the virtual table of any interface.

```
' QueryInterface method
' Returns a pointer to a specified interface on an object to
which a client currently holds an
' interface pointer. You must release the returned interface,
when no longer needed, with a call
' to the Release method.

FUNCTION IUnknown_QueryInterface (BYVAL pthis AS DWORD PTR,
BYREF riid AS GUID, BYREF ppvObj AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[0] USING IUnknown_QueryInterface(pthis,
riid, ppvObj) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' 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.

FUNCTION IUnknown_AddRef (BYVAL pthis AS DWORD PTR) AS DWORD
    LOCAL DWRESULT AS LONG
    CALL DWORD @@pthis[1] USING IUnknown_AddRef(pthis) TO
DWRESULT
    FUNCTION = DWRESULT
END FUNCTION

' Release method

' 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.

FUNCTION IUnknown_Release (BYVAL pthis AS DWORD PTR) AS DWORD
    LOCAL DWRESULT AS DWORD
    CALL DWORD @@pthis[2] USING IUnknown_Release(pthis) TO
DWRESULT
    FUNCTION = DWRESULT
END FUNCTION
```

QueryInterface: Navigating in an Object

Once you have an initial pointer to an interface on an object, COM 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—by telling 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 call QueryInterface to ask an object for an interface through which the client may invoke the desired operations.

Because 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. Suppose also that the object supports interface B but does not support interface C. The result is that 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. If the object refuses to provide the requested pointer, the client must be prepared to do without, either by not doing whatever it had intended to do with that object or by attempting to fall back on another, perhaps less powerful, interface. This feature of COM functionality 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, 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 legacy elements about new functionality is, surprisingly, a feature not present in most other object architectures.

Note: This is somewhat similar to the *GetProcAddress* function, used to retrieve the address of an exported function from an specified DLL. If *GetProcAddress* succeeds, we know that the function exists and we can safely call it using the returned address; if *QueryInterface* succeeds, we know that the interface exists and we can safely call his methods through the returned interface pointer.

Managing Object Lifetimes Through Reference Counting

In traditional object systems, the life cycle of objects—that is, the issues surrounding the creation and deletion of objects—is handled implicitly by the language (or the language run time) or explicitly by application programmers.

In an evolving, decentrally constructed system made up of reused components, it is no longer true that any client, or even any 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. (Readers familiar with garbage-collected programming environments, such as Java, may disagree; however, Java objects do not span machine or even process boundaries, and therefore the garbage collection is restricted to objects living within a single-process space. In addition, Java forces the use of a single programming language.) Even when the original client is done with the object, it cannot simply shut the object down, because some other client or clients might still have a reference to it.

One way to ensure that an object is no longer needed is to depend entirely on an underlying communication channel to inform the system when all connections to a cross-process or cross-channel object have disappeared. However, 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, 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. Programming languages such as Java, which inherently have their own lifetime management schemes, such as garbage collection, can use COM's reference counting to implement and use COM objects internally, allowing the programmer to avoid dealing with it.

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 do this only 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 appropriate 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 might 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 appropriate to deallocate an object is manual reference counting. Each object maintains a 32-bit (at least) 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 methods have the following simple implementations:

- AddRef increments the object's internal reference count.
- Release first decrements the object's internal reference count, and 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 methods (along with QueryInterface), which is shared between all interfaces, and therefore a reference count that applies to the entire object. However, from a client's perspective, reference counting is strictly and clearly a per-interface-pointer notion, and therefore objects that take advantage of this capability by dynamically constructing, destroying, loading, or unloading portions of their functionality based on the currently extant interface pointers may be implemented. These are colloquially called *tear-off* interfaces.

Whenever a client calls a method (or API function), such as QueryInterface, that returns a new interface pointer, 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 example 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 identifies an object to COM.) This allows the object implementation to manage resources efficiently.

Thread-safety is also an important issue in implementing AddRef and Release. For more information, see Processes, Apartments, and Threads.

Note: Threads are out of the scope of this tutorial. The above link gives access to the MSDN documentation about the subject.

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 one another in managing the object's lifetime. As long as the client object conforms to certain rules of use, the object, in effect, provides this management. These rules specify 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 residing within pointer variables that include all the internal computation state that holds an interface pointer. This would include variables in memory locations, in internal processor registers, and both programmer-generated 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 must be 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 because 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, and so on.

- 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 the 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, as follows:

- 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 to 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. Therefore, 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 object is known to contain the lifetime of the second, it is not necessary to have a reference count on the second object'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 preferred solution. COM's connectable objects architecture, for example, uses this approach.

When implementing or using reference-counted objects, it may be useful to apply *artificial reference counts*, which guarantee object stability during processing of a function. In implementing a method of an interface, you might 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 avoid this is to insert a call to AddRef at the beginning of the method implementation and pair it with a call to Release just before the method returns.

In some situations, the return values of AddRef and Release may be unstable and should not be relied upon; they should be used only for debugging or diagnostic purposes.

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 set of DLLs and EXEs (primarily ole32.dll and rpcss.exe) in Microsoft® Windows®, the COM library includes the following:

- 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 so 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 so 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 CoInitializeEx and also does some initialization required for compound documents. Therefore, threads that call OleInitialize cannot be free-threaded. For information on threading in clients and servers, refer to Processes, Apartments, 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.

Note The PowerBASIC compilers automatically handle the Initialization and Uninizialitation of the COM Library, so you don't normally have to call these functions in your code.

Single-threaded and multiple-threaded EXE modules that use COM can leave PowerBASIC to handle all initialization and uninitialization requirements for each thread. This also applies to DLLs created with the PowerBASIC for Windows compiler.

However, in some instances, it may be necessary to explicitly uninitialize the COM subsystem for a given thread. For example to force a COM Object to unload from memory when it is no longer in use. In such cases, an application may explicitly call the CoUninitialize API function within that thread. However, if COM services are subsequently required again by that same thread, the COM subsystem must be reinitialized by a call to CoInitialize.

[Excerpted from the PowerBASIC help file]

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, because it is more reasonable than defining case-by-case rules and so that the COM remote procedure call implementation can correctly manage memory.

The methods of a 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, and so forth.

The OLE Memory Allocator

The COM library provides an implementation of a memory allocator that is thread-safe. (That is, it cannot cause problems in multithreaded 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 COM 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. However, it is more efficient to call the helper functions CoTaskMemAlloc, CoTaskMemRealloc, 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.

The following wrapper functions allows you to use the IMalloc standard interface with the PowerBASIC compilers:

```
' Alloc method
' Allocates a block of memory.

FUNCTION IMalloc_Alloc (BYVAL pthis AS DWORD PTR, BYVAL cb AS
DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[3] USING IMalloc_Alloc(pthis, cb) TO
HRESULT
    FUNCTION = HRESULT
END FUNCTION

' Realloc method
' Changes the size of a previously allocated memory block.
```

```
FUNCTION IMalloc_Realloc (BYVAL pthis AS DWORD PTR, BYVAL pv AS
DWORD, BYVAL cb AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[4] USING IMalloc_Realloc(pthis, pv, cb)
TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' Free method
' Frees a previously allocated block of memory.

FUNCTION IMalloc_Free (BYVAL pthis AS DWORD PTR, BYVAL pv AS
DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[5] USING IMalloc_Free(pthis, pv) TO
HRESULT
    FUNCTION = HRESULT
END FUNCTION

' GetSize method
' Returns the size (in bytes) of a memory block previously
allocated with IMalloc::Alloc
' or IMalloc::Realloc. If pv is a null pointer the value
rerturned is -1.

FUNCTION IMalloc_GetSize (BYVAL pthis AS DWORD PTR, BYVAL pv AS
LONG) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[6] USING IMalloc_GetSize(pthis, pv) TO
HRESULT
    FUNCTION = HRESULT
END FUNCTION

' DidAlloc method
' Determines whether this allocator was used to allocate the
specified block of memory.

FUNCTION IMalloc_DidAlloc (BYVAL pthis AS DWORD PTR, BYVAL pv AS
DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[7] USING IMalloc_DidAlloc(pthis, pv) TO
HRESULT
    FUNCTION = HRESULT
END FUNCTION

' HeapMinimize method
' Minimizes the heap as much as possible by releasing unused
memory to the operating system,
' coalescing adjacent free blocks and committing free pages.
' Calling IMalloc::HeapMinimize is useful when an application
has been running for some time
' and the heap may be fragmented.

SUB IMalloc_HeapMinimize (BYVAL pthis AS DWORD PTR)
    CALL DWORD @@pthis[8] USING IMalloc_HeapMinimize(pthis)
END SUB
```

Memory Management Rules

The lifetime 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—Initially allocated by the caller, and then freed and reallocated 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 the following additional rules:

- In case of an error condition, parameters must always be reliably set to a value that will be cleaned up without any action by the caller.
- 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 members 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.

Note: Dual and dispatch interfaces usually use OLE strings (aka dynamic variable lenght strings), allocated with SysAllocateString and freed with SysFreeString, but the low-level COM methods found in the standard COM interfaces use CotaskMemAlloc and should be freed by CoTaskMemFree. Below is a wrapper function for the GetApplicationName method of the Itask property. The called method returns an Unicode Asciiz String and the function uses the API function lstrlenW to retrieve his lenght, PEEK$ to read the content, ACODE$ to convert it to Ascii and CoTaskMemFree to free the memory.

```
DECLARE FUNCTION Proto_ITask_GetApplicationName (BYVAL pthis AS
DWORD PTR, BYREF ppwszApplicationName AS DWORD) AS LONG

FUNCTION ITask_GetApplicationName (BYVAL pthis AS DWORD PTR,
BYREF strApplicationName AS STRING) AS LONG
    LOCAL HRESULT AS LONG
    LOCAL ppwszApplicationName AS DWORD PTR
    LOCAL bstrLen AS LONG
    LOCAL buffer AS STRING
    CALL DWORD @@pthis[33] USING
Proto_ITask_GetApplicationName(pthis, ppwszApplicationName) TO
HRESULT
    strApplicationName = ""
    IF ISTRUE ppwszApplicationName THEN
       bstrlen = lstrlenW(BYVAL ppwszApplicationName)
       IF ISTRUE bstrlen THEN
          buffer = PEEK$(ppwszApplicationName, bstrlen * 2)
          strApplicationName = ACODE$(buffer)
       END IF
       CoTaskMemFree ppwszApplicationName
    END IF
    FUNCTION = HRESULT
END FUNCTION
```

COM Clients and Servers

A critical aspect of COM is how clients and servers interact. 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. A COM server is any object that provides services to clients; these services are in the form of COM interface implementations that can be called by any client that 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 executable file (EXE). Out-of-process servers can reside either on the local machine or on a remote machine. In addition, COM provides a mechanism that allows an in-process server (a DLL) to run in a surrogate EXE process to gain the advantage of being able to run the process on a remote machine. For more information, see DLL Surrogates.

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 enables existing applications to interact with new applications and with each other across networks with proper administration, and new applications can be written to take advantage of networking features.

COM client applications do not need to be aware of how server objects are packaged, whether they are packaged as in-process objects (in DLLs) or as local or remote objects (in EXEs). Distributed COM further allows objects to be packaged as Microsoft Windows NT or Microsoft Windows 2000 Services, synchronizing COM with the rich administrative and system-integration capabilities of Windows NT and Windows 2000.

Note: Throughout this documentation the acronym *COM* is used in preference to *DCOM*. This is because DCOM is not separate—it is just COM with a longer wire. In cases where what is being described is specifically a remote operation, the term *distributed COM* is used.

COM is designed to make it possible to add the support for location transparency that extends across a network. It allows applications written for single machines to run across a network and provides features that extend these capabilities and add to the security necessary in a network. (For more information, see Security in COM.) COM specifies a mechanism by which the class code can be used by many different applications.

Getting a Pointer to an Object

Because COM does not have a strict class model, there are four ways that a client can instantiate or get a pointer to an interface on an object:

- Call a COM Library API function that creates an object of a predetermined type—that is, the function will return a pointer to only 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.

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

Creating an Object of a Predetermined Type

There are numerous COM functions, such as CoGetMalloc, that return pointers to specific interface implementations. (CoGetMalloc retrieves a pointer to the standard COM memory allocator.) Most of these are helper functions, and most of these functions are described in the reference sections of this documentation, under the specific area they are related to, such as storage or data transfer.

Creating an Object Based on a CLSID

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.

Returning a Pointer to a Separate 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. COM defines interfaces for enumerating a wide variety of items, such as strings, important structures, 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 COM 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.

Implementing an Object Through Which to Pass an Interface Pointer Directly to the Client

When two objects, such as an OLE compound document container and server, need bidirectional communication, each implements an object containing an interface method through which it can pass an interface 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 a client's ability to launch a server, create an instance of the server's object, and have access to the methods of the interfaces on the object.

COM provides extensions to this basic COM process that make it virtually seamless across a network. If a client is able to identify the server through its CLSID, calling a few simple functions permit COM to do all the work of locating and launching the server and activating the object. Subkeys in the registry 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, the COM object creation functions allow more flexibility and efficiency.

COM 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 Microsoft® Visual C++® class and a COM class: In Visual C++, a class is a type, while a COM class is simply a definition of the object and carries no type, although a C++ programmer might implement it by 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 executable application (EXE).

Each COM class is identified by a CLSID, a unique 128-bit GUID, which the server must register. COM 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 clients and servers on the same machine, 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 Microsoft Windows and Macintosh 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 CreateInstance method. You implement an IClassFactory interface for each class of object that you offer to be instantiated. (For more

information on implementing IClassFactory, see Implementing IClassFactory.)

Note: Servers that support some other custom class factory interface 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 be either 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.)

Note: COM has just a few API functions on which many of the others are built. The most important of these is probably CoGetClassObject, which underlies all of the instance creation functions.

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 that the client can do whatever work with the object that it wants to. Using COM's basic API functions, the client must also take care to release all object pointers.

Another mechanism for activating object instances is through the class moniker. Class monikers bind to the class object of the class for which they are created. For more information, see Class Monikers. COM provides several helper functions that reduce the work of creating object instances. These are described in Instance Creation Helper Functions.

The CreateObject helper function below creates an instance of an object passing either the ClsID (Class Identifier) or the ProgID (Programatic Identifier, an unique alphanumeric text string associated with a given COM object). It is a versatil function, since it allows to create instances of objects that don’t have a ProgID passing the ClsID, and can be used both with components that have dual interfaces (derived from Idispatch) and components whose interfaces derive directly from Iunknown. If it is a dispatch interface, a pointer to the IDispatch interface will be returned; otherwise, the function will return a pointer to the IUnknown interface.

```
' Creates an instance of an object.
' Parameters:
' strProgID
'   Required. String. The ProgID or the CLSID of the object to
create.
' ppv
'   Required. Dword. Address of pointer variable that receives
the interface pointer.
'   Upon failure, ppv contains %NULL.
' Return value:
'   One of the standard HRESULT values or %S_OK

FUNCTION CreateObject (BYVAL strProgID AS STRING, BYREF ppv AS
DWORD) EXPORT AS LONG

    LOCAL hr AS LONG                    ' HRESULT
    LOCAL pUnknown AS DWORD             ' IUnknown pointer
    LOCAL pDispatch AS DWORD            ' IDispatch pointer
    LOCAL IID_NULL AS GUID              ' Null GUID
    LOCAL IID_IUnknown AS GUID          ' Iunknown GUID
    LOCAL IID_IDispatch AS GUID         ' IDispatch GUID
    LOCAL ClassID AS GUID               ' CLSID

    ' Standard interface GUIDs
    IID_NULL = GUID$("{00000000-0000-0000-0000-000000000000}")
    IID_IUnknown = GUID$("{00000000-0000-0000-c000-
000000000046}")
    IID_IDispatch = GUID$("{00020400-0000-0000-c000-
000000000046}")

    ' Exit if strProgID is a null string
    IF strProgID = "" THEN
       FUNCTION = %E_INVALIDARG
       EXIT FUNCTION
    END IF

    ' Convert the ProgID in a CLSID
    ClassID = CLSID$(strProgID)

    ' If it fails, see if it is a CLSID
    IF ClassID = IID_NULL THEN ClassID = GUID$(strProgID)

    ' If not a valid ProgID or CLSID return an error
    IF ClassID = IID_NULL THEN
       FUNCTION = %E_INVALIDARG
       EXIT FUNCTION
    END IF

    ' Create an instance of the object
    ' Context: &H17 (%CLSCTX_ALL) =
    ' %CLSCTX_INPROC_SERVER OR %CLSCTX_INPROC_HANDLER OR _
    ' %CLSCTX_LOCAL_SERVER OR %CLSCTX_REMOTE_SERVER
    hr = CoCreateInstance(ClassID, BYVAL %NULL, &H17,
IID_IUnknown, pUnknown)
    IF ISTRUE hr OR ISFALSE pUnknown THEN
```

```
       FUNCTION = hr
       EXIT FUNCTION
    END IF

    ' Ask for the dispatch interface
    hr = IUnknown_QueryInterface(pUnknown, IID_IDispatch,
pDispatch)

    ' If it fails, return the Iunknown interface
    IF ISTRUE hr OR ISFALSE pDispatch THEN
       ppv = pUnknown
       FUNCTION = %S_OK
       EXIT FUNCTION
    END IF

    ' Release the IUnknown interface
    IUnknown_Release pUnknown

    ' Return a pointer to the dispatch interface
    ppv = pDispatch
    FUNCTION = %S_OK

END FUNCTION
```

The CreateControlLic uses CoGetClassObject and IClassFactory2 to create an unitialized instance of a licensed OCX.

```
DECLARE FUNCTION CoGetClassObject LIB "OLE32.DLL" ALIAS
"CoGetClassObject" (rclsid AS GUID, BYVAL dwclsContext AS DWORD,
BYVAL pServerInfo AS DWORD, riid AS GUID, ppv AS DWORD) AS DWORD

' IClassFactory2::CreateInstanceLic
' 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.

FUNCTION IClassFactory2_CreateInstanceLic (BYVAL pthis AS DWORD
PTR, BYVAL pUnkOuter AS DWORD, BYVAL pUnkReserved AS DWORD,
BYREF riid AS GUID, BYVAL pbstrKey AS DWORD, BYREF ppvObj AS
DWORD) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pthis[7] USING
IClassFactory2_CreateInstanceLic(pthis, pUnkOuter, pUnkReserved,
riid, pbStrKey, ppvObj) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' Creates an uninitialized instance of a licensed OCX.

FUNCTION CreateControlLic (BYVAL strProgID AS STRING, BYVAL
strLicKey AS STRING) AS LONG
```

```
    LOCAL HRESULT AS LONG               ' Result code
    LOCAL ppUnknown AS DWORD            ' IUnknown pointer
    LOCAL ppDispatch AS DWORD           ' IDispatch pointer
    LOCAL ppObj AS DWORD                ' Dispatch interface of
the control
    LOCAL ppClassFactory2 AS DWORD      ' IClassFactory2 pointer
    LOCAL ppUnkContainer AS DWORD       ' IUnknown of the
container
    LOCAL IID_NULL AS GUID              ' Null GUID
    LOCAL IID_IUnknown AS GUID          ' Iunknown GUID
    LOCAL IID_IDispatch AS GUID         ' IDispatch GUID
    LOCAL IID_IClassFactory2 AS GUID    ' IClassFactory2 GUID
    LOCAL ClassID AS GUID               ' CLSID
    LOCAL pbstrLicKey AS STRING         ' Unicode license key
string

    pbstrLicKey = UCODE$(strLicKey)     ' Convert the license
key to Unicode

    ' Standard interface GUIDs
    IID_NULL = GUID$("{00000000-0000-0000-0000-000000000000}")
    IID_IUnknown = GUID$("{00000000-0000-0000-c000-
000000000046}")
    IID_IDispatch = GUID$("{00020400-0000-0000-c000-
000000000046}")
    IID_IClassFactory2 = GUID$("{b196b28f-bab4-101a-b69c-
00aa00341d07}")

    ' Exit if strProgID is a null string
    IF strProgID = "" THEN
       FUNCTION = &H80070057 ' %E_INVALIDARG
       EXIT FUNCTION
    END IF

    ' Convert the ProgID to a CLSID
    ClassID = CLSID$(strProgID)

    ' If it fails, see if it is a CLSID
    IF ClassID = IID_NULL THEN ClassID = GUID$(strProgID)

    ' If not a valid ProgID or CLSID return an error
    IF ClassID = IID_NULL THEN
       FUNCTION = &H80070057 ' %E_INVALIDARG
       EXIT FUNCTION
    END IF

    ' Get a reference to the IClassFactory2 interface of the
control
    ' Context: &H17 (%CLSCTX_ALL) =
    ' %CLSCTX_INPROC_SERVER OR %CLSCTX_INPROC_HANDLER OR _
    ' %CLSCTX_LOCAL_SERVER OR %CLSCTX_REMOTE_SERVER
    HRESULT = CoGetClassObject(ClassID, &H17, %NULL,
IID_IClassFactory2, ppClassFactory2)
    IF ISTRUE HRESULT THEN
       FUNCTION = HRESULT
```

```
       EXIT FUNCTION
    END IF

    ' Create a licensed instance of the control
    HRESULT = IClassFactory2_CreateInstanceLic(ppClassFactory2,
%NULL, %NULL, IID_IUnknown, STRPTR(pbstrLicKey), ppUnknown)
    ' First release the IClassFactory2 interface
    IUnknown_Release ppClassFactory2
    IF ISTRUE HRESULT  OR ISFALSE ppUnknown THEN
       FUNCTION = HRESULT
       EXIT FUNCTION
    END IF

    ' Ask for the dispatch interface of the control
    HRESULT = IUnknown_QueryInterface(ppUnknown, IID_IDispatch,
ppDispatch)

    ' If it fails, use the IUnknown of the control, else use
IDispatch
    IF ISTRUE HRESULT OR ISFALSE ppDispatch THEN
       ppObj = ppUnknown
    ELSE
       ' Release the IUnknown interface
       IUnknown_Release ppUnknown
       ppObj = ppDispatch
    END IF

    FUNCTION = HRESULT

END FUNCTION
```

Locating a Remote Object

With the advent of COM for distributed systems, COM uses the basic model for object creation described in COM Class Objects and CLSIDs and adds more than one way to locate an object that might 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. Therefore, client applications 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 the CoGetClassObject 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.

Note: Instance creation calls, when they specify a server location, can override a registry setting. The algorithm COM uses for doing this is described in the reference for the CLSCTX enumeration.

Remote activation depends on the security relationship between client and server. For more information, see Security in COM.

Instance Creation Helper Functions

In previous releases of COM, 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 four important new instance creation mechanisms:

- Class Monikers and IClassActivator
- CoCreateInstanceEx
- CoGetInstanceFromFile
- CoGetInstanceFromIStorage

A class moniker permits you to identify the class of an object and is typically used with another moniker, like a file moniker, to indicate the location of the object. This permits you to bind to an object and specify the server that is to be launched for that object. Class monikers may also be composed to the right of monikers supporting binding to the IClassActivator interface. For more information, see Class Monikers.

Note: Monikers are out of the scope of this tutorial.

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 helper functions CoGetInstanceFromFile and CoGetInstanceFromIStorage encapsulate both the instance creation power of CoCreateInstanceEx and the initialization, the former from a file and the latter from a storage.

' 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.

```
TYPE MULTI_QI
   pIID AS GUID PTR
   pItf AS DWORD PTR
   hr AS LONG
END TYPE

' The COAUTHIDENTITY structure represents a user name and
password. A pointer to a  COAUTHIDENTITY structure is a member
of the COAUTHINFO structure, which specifies authentication
settings for remote activation requests.
TYPE COAUTHIDENTITY
   User AS ASCIIZ PTR
   UserLength AS DWORD
   Domain AS ASCIIZ PTR
   DomainLength AS DWORD
   Password AS ASCIIZ PTR
   PasswordLength AS DWORD
   Flags AS DWORD
END TYPE

' The COAUTHINFO structure specifies the authentication settings
used while making a remote activation request from the client
machine to the server machine.
TYPE COAUTHINFO
   dwAuthnSvc AS DWORD
   dwAuthzSvc AS DWORD
   pwszServerPrincName AS STRING PTR
   dwAuthnLevel AS DWORD
```

```
   dwImpersonationLevel AS DWORD
   pAuthIdentityData AS COAUTHIDENTITY PTR
   dwCapabilities AS DWORD
END TYPE

' Identifies a remote machine resource to the new or enhanced
activation functions.
TYPE COSERVERINFO
   dwReserved1 AS DWORD
   pwszName AS STRING PTR
   pAuthInfo AS COAUTHINFO PTR
   dwReserved2 AS DWORD
END TYPE

' Note: The last parameter of the following functions is a
pointer to an array of MULTI_QI
' structures. Pass it using the address of the first element of
the array obtained with VARPTR.

DECLARE FUNCTION  CoCreateInstanceEx LIB "OLE32.DLL" ALIAS
"CoCreateInstanceEx" ( _
   BYREF rclsid AS GUID, _
   BYREF pUnkOuter AS ANY, _
   BYVAL dwClsContext AS DWORD, _
   BYREF pServerInfo AS COSERVERINFO, _
   BYVAL cmq AS LONG, _
   BYVAL pResults AS DWORD _
) AS LONG

DECLARE FUNCTION  CoGetInstanceFromFile LIB "OLE32.DLL" ALIAS
"CoGetInstanceFromFile" ( _
   BYREF pServerInfo AS COSERVERINFO _
   BYREF pclsid AS GUID, _
   BYVAL punkPuter AS DWORD, _
   BYVAL dwClsCtx AS DWORD, _
   BYVAL grfMode AS DWORD, _
   BYVAL szName AS DWORD, _
   BYVAL cmq AS DWORD, _
   BYVAL rgmqResults AS DWORD _
) AS LONG

DECLARE FUNCTION  CoGetInstanceFromIStorage LIB "OLE32.DLL"
ALIAS "CoGetInstanceFromIStorage" ( _
   BYREF pServerInfo AS COSERVERINFO _
   BYREF pclsid AS GUID, _
   BYVAL punkPuter AS DWORD, _
   BYVAL dwClsCtx AS DWORD, _
   BYVAL pstg AS DWORD, _
   BYVAL cmq AS DWORD, _
   BYVAL rgmqResults AS DWORD _
) AS LONG
```

Error Handling

Almost all COM functions and interface methods return a value of the type HRESULT. The HRESULT (for *result handle*) is a way of returning success, warning, and error values. HRESULTs are really not handles to anything; they are only 32-bit values with several fields encoded in the value. As per the COM specification a result of zero 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 identical to an SCODE. COM uses only HRESULTs.

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.

Note: The Windows error codes are defined in Win32Api.inc. To use the equates you should include it in your application using the metastatment #INCLUDE “Win32Api.inc”.

The following function transforms a Windows error code in a localized error string:

```
FUNCTION SQLDMO_WinErrorMsg alias "SQLDMO_WinErrorMsg"
(BYVAL dwError AS DWORD) EXPORT AS STRING
   IF dwError = 0 THEN EXIT FUNCTION
   LOCAL pBuffer   AS ASCIIZ PTR
   LOCAL ncbBuffer AS DWORD
   ncbBuffer =
FormatMessage(%FORMAT_MESSAGE_ALLOCATE_BUFFER OR _
      %FORMAT_MESSAGE_FROM_SYSTEM OR
%FORMAT_MESSAGE_IGNORE_INSERTS, _
      BYVAL %NULL, dwError, BYVAL MAKELANGID(%LANG_NEUTRAL,
%SUBLANG_DEFAULT), _
      BYVAL VARPTR(pBuffer), 0, BYVAL %NULL)
   IF ncbBuffer THEN
      FUNCTION = @pBuffer
      LocalFree pBuffer
```

```
   END IF
END FUNCTION
```

Structure of COM 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 following illustration shows the format of an SCODE on a 16-bit platform; the numbers indicate bit positions:

HRESULTs and SCODEs on 32-bit platforms have the following format:

The high-order bit in the HRESULT or SCODE indicates whether the return value represents success or failure. If set to 0, SEVERITY_SUCCESS, the value indicates success. If set to 1, SEVERITY_ERROR, it indicates failure.

The context field is reserved in the SCODE on 16-bit platforms and does not exist in the version for 32-bit platforms. The R, C, N, and r bits are also reserved.

The facility field in both versions indicates the system service responsible for the error. Microsoft allocates new facility codes as they become necessary. Most SCODEs and HRESULTs set the facility field to FACILITY_ITF, indicating an interface method error.

Common facility fields are described in the following table.

Facility Field	Value	Description
%FACILITY_DISPATCH	2	For late-binding IDispatch interface errors.
%FACILITY_ITF	4	For most status codes returned from interface methods. The actual meaning of the error is defined by the interface. That is, two HRESULTs with exactly the same 32-bit value returned from two different interfaces might have different meanings.
%FACILITY_NULL	0	For broadly applicable common status codes such as S_OK.
%FACILITY_RPC	1	For status codes returned from remote procedure calls.
%FACILITY_STORAGE	3	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.
%FACILITY_WIN32	7	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.
%FACILITY_WINDOWS	8	Used for additional error codes from Microsoft-defined interfaces.

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 format:

Facility_Severity_Reason

Facility is either the facility name or some other distinguishing identifier; *Severity* is a single letter, S or E, that indicates whether the function call succeeded (S) or produced an error (E); 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. Microsoft reserves the right to define new failure codes (but not success codes) for the interfaces described in FACILITY_ITF or in new facilities.

Codes in FACILITY_ITF

HRESULTs 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 function or interface method from which they are returned. This means that 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 error code allocation system that avoids conflicting codes allocated by different programmers at different times in different places (unlike the handling of interface identifiers and CLSIDs). As a result, the 32-bit HRESULT is structured such that Microsoft can define several universal 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 be defined only by Microsoft.

2. Status codes in facility FACILITY_ITF are defined solely by the developer of the interface or function 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 COM-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 COM-defined errors.
It is also recommended that developers define new functions and interfaces to return error codes as defined by COM 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

COM 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 convert return values of different data types and are quite useful in code that runs only on 16-bit platforms, code that runs both on 16-bit and on 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 to provide compatibility and make porting easier. Newly written code should use the HRESULT macros and functions.

The error handling macros are described in the following table.

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 the severity bit, facility code, and error code that comprise the HRESULT.
MAKE_SCODE	Returns an SCODE given the severity bit, facility code, and error code that comprise the SCODE.
HRESULT_CODE	Extracts the error code part of the HRESULT.
HRESULT_FACILITY	Extracts the facility code 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 code from the SCODE.
SCODE_SEVERITY	Extracts the severity field from the SCODE.
SUCCEEDED	Tests the severity bit of the SCODE or HRESULT—returns TRUE if the severity is zero and FALSE if it is one.
FAILED	Tests the severity bit 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.
HRESULT_FROM_WIN32	Maps a Win32® error value into an HRESULT. This assumes that Win32 errors fall in the range -32K to 32K.
HRESULT_FROM_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 for a caller to know 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 documentation 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 because they are not meaningful, you do not need to write special code to handle them. A simple check for zero or nonzero is adequate.

HRESULTs

The return value of COM functions and methods is an HRESULT. The values of some HRESULTs have been changed in COM 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. Common HRESULTs and their 32-bit values are listed in the following table.

HRESULT	Value	Description
%E_ABORT	&H80004004	The operation was aborted because of an unspecified error.
%E_ACCESSDENIED	&H80070005	A general access-denied error.
%E_FAIL	&H80004005	An unspecified failure has occurred.
%E_HANDLE	&H80070006	An invalid handle was used.
%E_INVALIDARG	&H80070057	One or more arguments are invalid.
%E_NOINTERFACE	&H80004002	The QueryInterface method did not recognize the requested interface. The interface is not supported.
%E_NOTIMPL	&H80004001	The method is not implemented.
%E_OUTOFMEMORY	&H8007000E	The method failed to allocate necessary memory.
E_PENDING	&H8000000A	The data necessary to complete the operation is not yet available.
%E_POINTER	&H80004003	An invalid pointer was used.
%E_UNEXPECTED	&H8000FFFF	A catastrophic failure has occurred.

%S_FALSE	&H00000001	The method succeeded and returned the boolean value FALSE.
%S_OK	&H00000000	The method succeeded. If a boolean return value is expected, the returned value is TRUE.

Requirements

With the PowerBASIC compilers, to use the return values, you must #INCLUDE “Win32Api.inc”.

Win32 and Network Errors

If the first four digits of the error code are 8007, this indicates a Win32 or network error. You can use the net command to decode these types of errors. To decode the error, first convert the last four digits of the hexadecimal error code to decimal. Then, at the command prompt, type the following, where *decimal code* is replaced with the return value you want to decode:

net helpmsg <*decimal code*>

The net command returns a description of the error. For example, if COM returns the error 8007054B, convert the 054B to decimal (1355). Then type the following:

net helpmsg 1355

The net command returns the error description:

The specified domain did not exist.

Handling Unknown Errors

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. Pay particular attention to this potential problem when propagating error codes from functions that are called internally.

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 COM-defined interfaces and functions. 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.
It is important to handle all possible errors when calling an interface method. Failure to do so could cause your application to crash, to corrupt data, or to become vulnerable to security exploits. The following code sample shows the recommended way of handling unknown errors:

```
LOCAL hr AS LONG

hr = xxMethod()

SELECT CASE hr

   CASE %NOERROR
      ‘ Method returned success.

   CASE x1
      ‘ Handle error x1 here.

   CASE x2
      ‘ Handle error x2 here.

   CASE %E_UNEXPECTED
      ‘ Handle unexpected errors here.

END SELECT
```

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

```
IF xxMethod() = %NOERROR THEN
    ‘ Handle success here.
ELSE
    ‘ Handle failure here.
END IF
```

Note: Some components, such SQLDMO, return extended error information through the error handling interfaces IErrorInfo, ICreateErrorInfo and ISupportErrorInfo.

The following wrapper functions allow you to use these interfaces with the PowerBASIC compilers:

```
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.

' Indicates whether an interface supports the IErrorInfo
interface.
FUNCTION ISupportErrorInfo_InterfaceSupportsErrorInfo (BYVAL
pErrorInfo AS DWORD PTR, BYREF riid AS GUID) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[3] USING
ISupportErrorInfo_InterfaceSupportsErrorInfo (pErrorInfo, riid)
TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

The ICreateErrorInfo interface returns error information.

' Sets the GUID for the interface that defined the error.
FUNCTION ICreateErrorInfo_SetGUID (BYVAL pErrorInfo AS DWORD
PTR, BYREF rguid AS GUID) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[3] USING ICreateErrorInfo_SetGUID
(pErrorInfo, rguid) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' Sets the ProgID for the class or application that returned the
error.
DECLARE FUNCTION Proto_ICreateErrorInfo_SetSource (BYVAL
pErrorInfo AS DWORD PTR, BYVAL szSource AS DWORD) AS LONG

FUNCTION ICreateErrorInfo_SetSource (BYVAL pErrorInfo AS DWORD
PTR, BYVAL strSource AS STRING) AS LONG
   LOCAL HRESULT AS LONG
   strSource = UCODE$(strSource)
   CALL DWORD @@pErrorInfo[4] USING
Proto_ICreateErrorInfo_SetSource (pErrorInfo, STRPTR(strSource))
TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' Sets a textual description of the error.
DECLARE FUNCTION Proto_ICreateErrorInfo_SetDescription (BYVAL
pErrorInfo AS DWORD PTR, BYVAL szDescription AS DWORD)  AS LONG

FUNCTION ICreateErrorInfo_SetDescription (BYVAL pErrorInfo AS
DWORD PTR, BYVAL strDescription AS STRING) AS LONG
   LOCAL HRESULT AS LONG
   strDescription = UCODE$(strDescription)
   CALL DWORD @@pErrorInfo[5] USING
Proto_ICreateErrorInfo_SetDescription (pErrorInfo,
STRPTR(strDescription)) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' Sets the path of the Help file that describes the error.
```

```
DECLARE FUNCTION Proto_ICreateErrorInfo_SetHelpFile (BYVAL
pErrorInfo AS DWORD PTR, BYVAL szHelpFile AS DWORD) AS LONG

FUNCTION ICreateErrorInfo_SetHelpFile (BYVAL pErrorInfo AS DWORD
PTR, BYVAL strHelpFile AS STRING) AS LONG
   LOCAL HRESULT AS LONG
   strHelpFile = UCODE$(strHelpFile)
   CALL DWORD @@pErrorInfo[6] USING
Proto_ICreateErrorInfo_SetHelpFile (pErrorInfo,
STRPTR(strHelpFile)) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' Sets the Help context ID for the error.
FUNCTION  ICreateErrorInfo_SetHelpContext (BYVAL pErrorInfo AS
DWORD PTR, BYVAL dwHelpContext AS DWORD)  AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[7] USING
ICreateErrorInfo_SetHelpContext (pErrorInfo, dwHelpContext) TO
HRESULT
   FUNCTION = HRESULT
END FUNCTION


The IErrorInfo interface provides detailed contextual error
information.

' Returns the globally unique identifier (GUID) for the
interface that defined the error.
FUNCTION IErrorInfo_GetGUID (BYVAL pErrorInfo AS DWORD PTR,
BYREF pguid AS GUID) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[3] USING IErrorInfo_GetGUID
(pErrorInfo, pguid) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' Returns the programmatic identifier (ProgID) for the class or
application that returned the error.
FUNCTION IErrorInfo_GetSource (BYVAL pErrorInfo AS DWORD PTR,
BYREF strSource AS STRING) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[4] USING IErrorInfo_GetSource
(pErrorInfo, strSource) TO HRESULT
   strSource = ACODE$(strSource)
   FUNCTION = HRESULT
END FUNCTION

' Returns a textual description of the error.
FUNCTION IErrorInfo_GetDescription (BYVAL pErrorInfo AS DWORD
PTR, BYREF strDescription AS STRING) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[5] USING IErrorInfo_GetDescription
(pErrorInfo, strDescription) TO HRESULT
   strDescription = ACODE$(strDescription)
   FUNCTION = HRESULT
```

```
END FUNCTION

' Returns the path of the Help file that describes the error.
FUNCTION IErrorInfo_GetHelpFile (BYVAL pErrorInfo AS DWORD PTR,
BYREF strHelpFile AS STRING)  AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[6] USING IErrorInfo_GetHelpFile
(pErrorInfo, strHelpFile) TO HRESULT
   strHelpFile = ACODE$(strHelpFile)
   FUNCTION = HRESULT
END FUNCTION

' Returns the Help context identifier (ID) for the error.
FUNCTION IErrorInfo_GetHelpContext (BYVAL pErrorInfo AS DWORD
PTR, BYREF pdwHelpContext AS DWORD)  AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pErrorInfo[7] USING IErrorInfo_GetHelpContext
(pErrorInfo, pdwHelpContext) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION
```

To retrieve extended error information (if the component that you're using supports it), call GetErrorInfo as soon as possible after the called COM function that has produced the error returns.

```
LOCAL hr AS LONG
LOCAL pErrorInfo AS DWORD
LOCAL strErrDesc AS STRING

hr = GetErrorInfo(BYVAL 0, pErrorInfo)
IF hr = %S_OK THEN
   hr = IErrorInfo_GetDescription(pErrorInfo, strErrDesc)
END IF
```

Events in COM and Connectable Objects

When a program detects something that has happened, it can notify its clients. For example, if a stock ticker program detects a change in the price of a stock, it can notify all clients of the change. This notification process is referred to as firing an event.

With COM, server objects can use COM events to fire events without any information about what objects will be notified. Objects can also use *connectable objects* to maintain detailed information about clients who have requested notifications.

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 bidirectional 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, 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 and 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 is only one piece of the overall architecture of connectable objects. This technology includes the following elements:

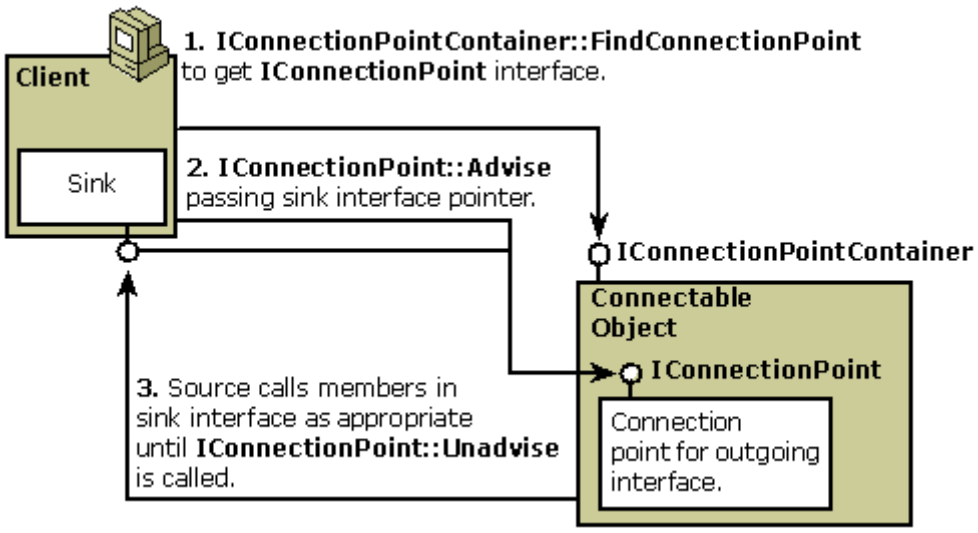
- Connectable object Implements the IConnectionPointContainer interface; creates at least one connection point object; defines an outgoing interface for the client.

- Client Queries the object for IConnectionPointContainer to determine whether 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:



Before the connection point object calls methods in the sink interface in step 3 in the preceding diagram, 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 multicasting.

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 as follows:

1. The client queries for IConnectionPointContainer on the object to determine whether 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 in IConnectionPointContainer::EnumConnectionPoints. There are a few additional steps needed if EnumConnectionPoints is used. (See Using IConnectionPointContainer 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, and 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, therefore, 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 that 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.

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 IConnectionPointContainer::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. Basically, 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.

After 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 by 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. (See Using IProvideClassInfo.)

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 either through the enumerator or through FindConnectionPoint.

Using IconnectionPoint

When the client has a pointer to a connection point, it can perform the following operations as expressed through IConnectionPoint:

- First, IConnectionPoint::GetConnectionInterface 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 IConnectionPoint::GetConnectionPointContainer method.

- Third, the most interesting methods for the client are IConnectionPoint::Advise and IConnectionPoint::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 to it that exist through IConnectionPoint::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.

The code below illustrates how to use the IconnectionPointContainer and IconnectionPoint interfaces to sink to the events fired by a component through a dispatch interface. It retrieves a connection pointer calling the FindConnectionPoint method, builds an Idispatch virtual table (an array of pointers to the implemented seven dispatch functions) and calls the Advise method to pass to the component the address of this virtual table. The component will use these addresses to call our implemented functions.

```
' ConnectionEvents dispatch interface
' IID = {00000400-0000-0010-8000-00AA006D2EA4}
' Attributes = 4096 [&H1000]  [Dispatchable]
' Number of functions = 9

TYPE ADODBConnectionEvents_EXCEPINFO
   wCode AS WORD
   wReserved AS WORD
   bstrSource AS DWORD
   bstrDescription AS DWORD
   bstrHelpFile AS DWORD
   dwHelpContext AS DWORD
   pvReserved AS DWORD
   pfnDeferredFillIn AS DWORD
   scode AS DWORD
END TYPE

FUNCTION ADODBConnectionEvents_IUnknown_QueryInterface (BYVAL
pthis AS DWORD PTR, _
   BYREF riid AS GUID, BYREF ppvObj AS DWORD) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pthis[0] USING
ADODBConnectionEvents_IUnknown_QueryInterface(pthis, riid,
ppvObj) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

FUNCTION ADODBConnectionEvents_IUnknown_Release (BYVAL pthis AS
DWORD PTR) AS DWORD
   LOCAL DWRESULT AS DWORD
   CALL DWORD @@pthis[2] USING
ADODBConnectionEvents_IUnknown_Release(pthis) TO DWRESULT
   FUNCTION = DWRESULT
END FUNCTION

' IConnectionPointContainer::FindConnectionPoint
' Returns a pointer to the IConnectionPoint interface of a
connection point for a specified IID,
' if that IID describes a supported outgoing interface.

FUNCTION
ADODBConnectionEvents_IConnectionPointContainer_FindConnectionPo
int ( _
   BYVAL pthis AS DWORD PTR, BYREF riid AS GUID, BYREF ppCP AS
DWORD) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pthis[4] USING
ADODBConnectionEvents_IConnectionPointContainer_FindConnectionPo
int(pthis, riid, ppCP) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' IConnectionPoint::Advise
' Establishes a connection between the connection point object
and the client's sink.
```

```
FUNCTION ADODBConnectionEvents_IConnectionPoint_Advise (BYVAL
pthis AS DWORD PTR, _
   BYVAL pUnkSink AS DWORD, BYREF pdwCookie AS DWORD) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pthis[5] USING
ADODBConnectionEvents_IConnectionPoint_Advise(pthis, pUnkSink,
pdwCookie) TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' IConnectionPoint::Unadvise
' Terminates an advisory connection previously established
through IConnectionPoint_Advise.
' The dwCookie parameter identifies the connection to terminate.

FUNCTION ADODBConnectionEvents_IConnectionPoint_Unadvise (BYVAL
pthis AS DWORD PTR, BYVAL dwCookie AS DWORD) AS LONG
   LOCAL HRESULT AS LONG
   CALL DWORD @@pthis[6] USING
ADODBConnectionEvents_IConnectionPoint_Unadvise(pthis, dwCookie)
TO HRESULT
   FUNCTION = HRESULT
END FUNCTION

' IDispatch virtual table

TYPE ADODBConnectionEvents_IDispatchVtbl
   QueryInterface AS DWORD     ' Returns pointers to supported
interfaces
   AddRef AS DWORD             ' Increments reference count
   Release AS DWORD            ' Decrements reference count
   GetTypeInfoCount AS DWORD   ' Retrieves the number of type
descriptions
   GetTypeInfo AS DWORD        ' Retrieves a description of
object's programmable interface
   GetIDsOfNames AS DWORD      ' Maps name of method or property
to DispId
   Invoke AS DWORD             ' Calls one of the object's
methods, or gets/sets one of its properties
   pVtblAddr AS DWORD          ' Address of the virtual table
   cRef AS DWORD               ' Reference counter
   pthis AS DWORD              ' IUnknown or IDispatch of the
control that fires the events
END TYPE

FUNCTION ADODBConnectionEvents_AddRef (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR) AS DWORD
   INCR @@pCookie.cRef
   FUNCTION = @@pCookie.cRef
END FUNCTION

FUNCTION ADODBConnectionEvents_QueryInterface (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR, _
   BYREF riid AS GUID, BYREF ppvObj AS DWORD) AS LONG
   ppvObj = pCookie
   ADODBConnectionEvents_AddRef pCookie
```

```
   FUNCTION = %S_OK
END FUNCTION

FUNCTION ADODBConnectionEvents_Release (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR) AS DWORD
   LOCAL pVtblAddr AS DWORD
   IF @@pCookie.cRef = 1 THEN
      pVtblAddr = @@pCookie.pVtblAddr
      IF ISFALSE HeapFree(GetProcessHeap(), 0, BYVAL pVtblAddr)
THEN
         FUNCTION = @@pCookie.cRef
         EXIT FUNCTION
      END IF
   END IF
   DECR @@pCookie.cRef
   FUNCTION = @@pCookie.cRef
END FUNCTION

FUNCTION ADODBConnectionEvents_GetTypeInfoCount (BYVAL pCookie
AS ADODBConnectionEvents_IDispatchVtbl PTR, BYREF pctInfo AS
DWORD) AS LONG
   FUNCTION = %E_NOTIMPL
END FUNCTION

FUNCTION ADODBConnectionEvents_GetTypeInfo (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR, _
   BYVAL itinfo AS DWORD, BYVAL lcid AS DWORD, BYREF pptinfo AS
DWORD) AS LONG
   FUNCTION = %E_NOTIMPL
END FUNCTION

FUNCTION ADODBConnectionEvents_GetIDsOfNames ( BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR, _
   BYREF riid AS GUID, BYVAL rgszNames AS DWORD, BYVAL cNames AS
DWORD, BYVAL lcid AS DWORD, BYREF rgdispid AS LONG) AS LONG
   FUNCTION = %E_NOTIMPL
END FUNCTION

' Builds the IDispatch Virtual Table

FUNCTION ADODBConnectionEvents_BuildVtbl (BYVAL pthis AS DWORD)
AS DWORD

   LOCAL pVtbl AS ADODBConnectionEvents_IDispatchVtbl PTR
   LOCAL pUnk AS ADODBConnectionEvents_IDispatchVtbl PTR

   pVtbl = HeapAlloc(GetProcessHeap(), %HEAP_ZERO_MEMORY,
SIZEOF(@pVtbl))
   IF pVtbl = 0 THEN EXIT FUNCTION

   @pVtbl.QueryInterface   =
CODEPTR(ADODBConnectionEvents_QueryInterface)
   @pVtbl.AddRef           =
CODEPTR(ADODBConnectionEvents_AddRef)
   @pVtbl.Release          =
CODEPTR(ADODBConnectionEvents_Release)
```

```
   @pVtbl.GetTypeInfoCount =
CODEPTR(ADODBConnectionEvents_GetTypeInfoCount)
   @pVtbl.GetTypeInfo      =
CODEPTR(ADODBConnectionEvents_GetTypeInfo)
   @pVtbl.GetIDsOfNames    =
CODEPTR(ADODBConnectionEvents_GetIDsOfNames)
   @pVtbl.Invoke           =
CODEPTR(ADODBConnectionEvents_Invoke)
   @pVtbl.pVtblAddr        = pVtbl
   @pVtbl.pthis            = pthis

   pUnk = VARPTR(@pVtbl.pVtblAddr)
   FUNCTION = pUnk

END FUNCTION

' Establishes a connection between the connection point object
and the client's sink.
' Returns a token that uniquely identifies this connection.

FUNCTION ADODBConnectionEvents_ConnectEvents (BYVAL pthis AS
DWORD, BYREF pdwCookie AS DWORD) AS LONG

   LOCAL HRESULT AS LONG                ' HRESULT code
   LOCAL pCPC AS DWORD                  '
IConnectionPointContainer
   LOCAL pCP AS DWORD                   ' IConnectionPoint
   LOCAL IID_CPC AS GUID                '
IID_IConnectionPointContainer
   LOCAL IID_CP AS GUID                 ' Events dispinterface
   LOCAL dwCookie AS DWORD              ' Returned token
   LOCAL pUnkSink AS DWORD              ' IUnknown of the class

   IID_CPC = GUID$("{B196B284-BAB4-101A-B69C-00AA00341D07}")
   IID_CP  = GUID$("{00000400-0000-0010-8000-00AA006D2EA4}")

   IF pthis = 0 THEN FUNCTION = -1 : EXIT FUNCTION
   HRESULT =
ADODBConnectionEvents_IUnknown_QueryInterface(pthis, IID_CPC,
pCPC)
   IF HRESULT <> %S_OK THEN FUNCTION = HRESULT : EXIT FUNCTION

   HRESULT =
ADODBConnectionEvents_IConnectionPointContainer_FindConnectionPo
int(pCPC, IID_CP, pCP)
   ADODBConnectionEvents_IUnknown_Release pCPC
   IF HRESULT <> %S_OK THEN FUNCTION = HRESULT : EXIT FUNCTION

   pUnkSink = ADODBConnectionEvents_BuildVtbl(pthis)
   IF ISTRUE pUnkSink THEN HRESULT =
ADODBConnectionEvents_IConnectionPoint_Advise(pCP, pUnkSink,
dwCookie)
   ADODBConnectionEvents_IUnknown_Release pCP
   pdwCookie = dwCookie
   FUNCTION = HRESULT
```

```
END FUNCTION

' Releases the events connection identified with the cookie
returned by the ConnectEvents function

FUNCTION ADODBConnectionEvents_DisconnectEvents (BYVAL pthis AS
DWORD, BYVAL dwCookie AS DWORD) AS LONG

   LOCAL HRESULT AS LONG                ' HRESULT code
   LOCAL pCPC AS DWORD                  '
IConnectionPointContainer
   LOCAL pCP AS DWORD                   ' IConnectionPoint
   LOCAL IID_CPC AS GUID                '
IID_IConnectionPointContainer
   LOCAL IID_CP AS GUID                 ' ConnectionEvents
dispinterface

   IID_CPC = GUID$("{B196B284-BAB4-101A-B69C-00AA00341D07}")
   IID_CP  = GUID$("{00000400-0000-0010-8000-00AA006D2EA4}")

   IF pthis = 0 THEN FUNCTION = -1 : EXIT FUNCTION
   HRESULT =
ADODBConnectionEvents_IUnknown_QueryInterface(pthis, IID_CPC,
pCPC)
   IF HRESULT <> %S_OK THEN FUNCTION = HRESULT : EXIT FUNCTION

   HRESULT =
ADODBConnectionEvents_IConnectionPointContainer_FindConnectionPo
int(pCPC, IID_CP, pCP)
   ADODBConnectionEvents_IUnknown_Release pCPC
   IF HRESULT <> %S_OK THEN FUNCTION = HRESULT : EXIT FUNCTION

   HRESULT =
ADODBConnectionEvents_IConnectionPoint_Unadvise(pCP, dwCookie)
   ADODBConnectionEvents_IUnknown_Release pCP
   FUNCTION = HRESULT

END FUNCTION

' Handles the events

FUNCTION ADODBConnectionEvents_Invoke (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR, BYVAL dispidMember AS
LONG, BYREF riid AS GUID, _
   BYVAL lcid AS DWORD, BYVAL wFlags AS WORD, BYREF pdispparams
AS DISPPARAMS, BYREF pvarResult AS VARIANT, _
   BYREF pexcepinfo AS ADODBConnectionEvents_EXCEPINFO, BYREF
puArgErr AS DWORD) AS LONG

   FUNCTION = %S_OK

   IF VARPTR(pdispparams) THEN

      SELECT CASE AS LONG dispidMember

         CASE &H00000000  ' (0)  ' // InfoMessage
```

```
            ' Handle the InfoMessage event
         CASE &H00000001  ' (1)  ' // BeginTransComplete
            ' Handle the BeginTransComplete event
         CASE &H00000003  ' (3)  ' // CommitTransComplete
            ' Handle the CommitTransComplete event
         CASE &H00000002  ' (2)  ' // RollbackTransComplete
            ' Handle the RollbackTransComplete event
         CASE &H00000004  ' (4)  ' // WillExecute
            ' Handle the WillExecute event
         CASE &H00000005  ' (5)  ' // ExecuteComplete
            ' Handle the ExecuteComplete event
         CASE &H00000006  ' (6)  ' // WillConnect
            ' Handle the WillConnect event
         CASE &H00000007  ' (7)  ' // ConnectComplete
            ' Handle the ConnectComplete event
         CASE &H00000008  ' (8)  ' // Disconnect
            ' Handle the Disconnect event

      END SELECT

   END IF

END FUNCTION
```

It also illustrates in his simplest form how the QueryInterface and Release methods work. QueryInterface returns the address of a pointer to our virtual table and calls AddRef to increment the reference count; Release decrements the reference count and frees the memory used by our class when this count reaches 0.

```
FUNCTION ADODBConnectionEvents_AddRef (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR) AS DWORD
   INCR @@pCookie.cRef
   FUNCTION = @@pCookie.cRef
END FUNCTION

FUNCTION ADODBConnectionEvents_QueryInterface (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR, _
   BYREF riid AS GUID, BYREF ppvObj AS DWORD) AS LONG
   ppvObj = pCookie
   ADODBConnectionEvents_AddRef pCookie
   FUNCTION = %S_OK
END FUNCTION

FUNCTION ADODBConnectionEvents_Release (BYVAL pCookie AS
ADODBConnectionEvents_IDispatchVtbl PTR) AS DWORD
   LOCAL pVtblAddr AS DWORD
   IF @@pCookie.cRef = 1 THEN
      pVtblAddr = @@pCookie.pVtblAddr
      IF ISFALSE HeapFree(GetProcessHeap(), 0, BYVAL pVtblAddr)
THEN
         FUNCTION = @@pCookie.cRef
         EXIT FUNCTION
      END IF
   END IF
```

```
   DECR @@pCookie.cRef
   FUNCTION = @@pCookie.cRef
END FUNCTION
```

Using IprovideClassInfo

A connectable object can offer the IProvideClassInfo and IProvideClassInfo2 interfaces so that its clients can easily examine its type 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 that it can dynamically provide a sink supporting the right entry points, as follows:

1. The client enumerates the connection points and then, to obtain the IIDs of outgoing interfaces supported by the connectable object, calls IConnectionPoint::GetConnectionInterface for each connection point.

2. The client queries the connectable object for one of the IProvideClassInfo interfaces.

3. The client calls methods in the IProvideClassInfo interfaces to get the type information for the outgoing interface.

4. The client creates a sink object supporting the outgoing interface.

5. The process continues, and the client calls 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.

The following wrapper functions allows you to use the IProvideClassInfo and IProvideClassInfo2 interfaces:

```
' IProvideClassInfo
' IID: {B196B283-BAB4-101A-B69C-00AA00341D07}
```

```
' GetClassInfo method
' 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.

FUNCTION IProvideClassInfo_GetClassInfo (BYVAL pthis AS DWORD
PTR, BYREF ppTI AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[3] USING
IProvideClassInfo_GetClassInfo(pthis, ppTI) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' IProvideClassInfo2
' IID: {B196B283-BAB4-101A-B69C-00AA00341D07}

' The GUIDKIND enumeration values are flags used to specify the
kind of information requested from an object in the
IProvideClassInfo2.

%GUIDKIND_DEFAULT_SOURCE_DISP_IID    = 1,
' The interface identifier (IID) of the object's outgoing
dispinterface, labeled [source, default]. The outgoing interface
in question must be derived from IDispatch.


' GetGUID method
' 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.

FUNCTION IProvideClassInfo2_GetGUID (BYVAL pthis AS DWORD PTR,
BYVAL dwGuidKind AS DWORD, BYREF pGUID AS GUID) AS LONG

    LOCAL HRESULT AS DWORD
    CALL DWORD @@pthis[4] USING
IProvideClassInfo2_GetGUID(pthis, dwGuidKind, pGUID) TO HRESULT
    FUNCTION = HRESULT

END FUNCTION
```

Enumerating Collections

To allow you to enumerate the number of items of a given type that an object maintains, COM 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 enumerator 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 COM 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 that contain advisory connection information for a data object.
IEnumSTATSTG	An array of STATSTG structures that 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.

The 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.

The code below implements a generic collection enumerator. For big collections, it is much faster than the technique suggested in the PowerBASIC help file. It also allows to enumerate collections that don't support an index t the *Item* method or that expect a key instead of an index, like the crazy Item method of the the Microsoft Scripting Object *Drives*' collection, that expects that you pass the name of a drive to return an object that allows you to get the name of that drive!

```
' ENUMERATOR
'
' Enumerates a collection and returns its contents in an array
of variants.
' This is a generic enumerator, so you don't need to select any
typelib to generate it.
' Collections are in general enumerated using the Item property
of the interfaces, but in
' many cases, it expects a key instead of am index. In the case
of the FileSystemObject, to
' enumerate the Files collection you will need to pass the name
of a file to get a reference
' that allows to get the name of this file! In such a situation,
we need a generic enumerator,
' such the ForEach function of Visual Basic. The following
wrapper functions provide access to
' the IEnumVARIANT interface, and an example of how to use it.

' EXCEPINFO structure

TYPE TB_Collection_EXCEPINFO
   wCode AS WORD               ' An error code describing the
error.
   wReserved AS WORD           ' Reserved
   bstrSource AS DWORD         ' Source of the exception.
   bstrDescription AS DWORD    ' Textual description of the
error.
   bstrHelpFile AS DWORD       ' Help file path.
   dwHelpContext AS DWORD      ' Help context ID.
   pvReserved AS DWORD         ' Reserved.
   pfnDeferredFillIn AS DWORD  ' Pointer to function that fills
in Help and description info.
   scode AS DWORD              ' An error code describing the
error.
END TYPE

' 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.

FUNCTION TB_Collection_IUnknown_QueryInterface (BYVAL pthis AS
DWORD PTR, BYREF riid AS GUID, BYREF ppvObj AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
```

```
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[0] USING
TB_Collection_IUnknown_QueryInterface(pthis, riid, ppvObj) TO
HRESULT
    FUNCTION = HRESULT
END FUNCTION

' 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.

FUNCTION TB_Collection_IUnknown_AddRef (BYVAL pthis AS DWORD
PTR) AS DWORD
    LOCAL DWRESULT AS LONG
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[1] USING
TB_Collection_IUnknown_AddRef(pthis) TO DWRESULT
    FUNCTION = DWRESULT
END FUNCTION

' 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.

FUNCTION TB_Collection_IUnknown_Release (BYVAL pthis AS DWORD
PTR) AS DWORD
    LOCAL DWRESULT AS DWORD
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[2] USING
TB_Collection_IUnknown_Release(pthis) TO DWRESULT
    FUNCTION = DWRESULT
END FUNCTION

' Provides access to properties and methods exposed by an
object.

FUNCTION TB_Collection_IDispatch_Invoke (BYVAL pthis AS DWORD
PTR, BYVAL dispidMember AS LONG, BYREF riid AS GUID, _
    BYVAL lcid AS DWORD, BYVAL wFlags AS WORD, BYREF pdispparams
AS DISPPARAMS, BYREF pvarResult AS VARIANT, _
    BYREF pexcepinfo AS TB_Collection_EXCEPINFO, BYREF puArgErr
AS DWORD) AS LONG

    LOCAL HRESULT AS LONG
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[6] USING
TB_Collection_IDispatch_Invoke(pthis, dispidMember, riid, lcid,
wFlags, pdispparams, pvarResult, pexcepinfo, puArgErr) TO
HRESULT
    FUNCTION = HRESULT
```

```
END FUNCTION

' HRESULT Next([in] UI4 celt, [in] *VARIANT rgvar, [out] *UI4
pceltFetched)
' The Next method enumerates the next celt elements in the
enumerator's list, returning them in
' rgelt along with the actual number of enumerated elements in
pceltFetched.
' Parameters:
' celt
'   [in] Number of items in the array.
' rgelt
'   [out] Address of array containing items.
' pceltFetched
'   [out] Address of variable containing actual number of items.
' Return Value:
'   Returns %S_OK if the method succeeds.

FUNCTION TB_Collection_IEnumVARIANT_Next (BYVAL pthis AS DWORD
PTR, BYVAL celt AS DWORD, BYVAL rgelt AS DWORD, BYREF
pceltFetched AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[3] USING
TB_Collection_IEnumVARIANT_Next(pthis, celt, rgelt,
pceltFetched) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' HRESULT Skip([in] UI4 celt)
' The Skip method Instructs the enumerator to skip the next celt
elements in the enumeration so
' the next call to TB_Collection_IEnumVARIANT_Next does not
return those elements.
' Parameter:
' celt
'   [in] Number of items to skip.
' Return Value:
'   Returns %S_OK if the method succeeds.

FUNCTION TB_Collection_IEnumVARIANT_Skip (BYVAL pthis AS DWORD
PTR, BYVAL celt AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[4] USING
TB_Collection_IEnumVARIANT_Skip(pthis, celt) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' HRESULT Reset()
' The Reset method instructs the enumerator to position itself
at the beginning of the list
' of elements.
' Return Value:
```

```
'   Returns %S_OK if the method succeeds.

FUNCTION TB_Collection_IEnumVARIANT_Reset (BYVAL pthis AS DWORD
PTR) AS LONG
    LOCAL HRESULT AS LONG
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[5] USING
TB_Collection_IEnumVARIANT_Reset(pthis) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' HRESULT Clone([out] **IEnumVARIANT ppenum)
' The Clone method creates another items 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.
' Parameters:
' ppenum
'   [out] Address of a variable that receives the IEnumVARIANT
interface pointer.
' Return Value:
'   Returns %S_OK if the method succeeds.
' Remarks
'   The caller must release the new enumerator separately from
the first enumerator.

FUNCTION TB_Collection_IEnumVARIANT_Clone (BYVAL pthis AS DWORD
PTR, BYVAL ppenum AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    CALL DWORD @@pthis[6] USING
TB_Collection_IEnumVARIANT_Clone(pthis, ppenum) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' _NewEnum method  [Restricted]
' Member identifier: -4 (_NewEnum always has a DispID of -4)
' Returns a a reference to the IUnknown interface of a
collection.
' Notes: Some objects declare it as a method and others as a
property, so we are passing
' both flags, %DISPATH_METHOD (=1) and %DISPATCH_PROPERTYGET
(=2) = 1 OR 2.
' The _NewEnum property can have a different VTable offset for
each collection. Since we want
' a generic function, we have to call Invoke taking advantage of
the fact that _NewEnum has
' always a DispID of -4.

FUNCTION TB_Collection_NewEnum (BYVAL pthis AS DWORD, BYREF
ppenum AS DWORD) AS DWORD
    DIM IID_NULL AS GUID
```

```
    DIM uDispParams AS DISPPARAMS
    DIM vResult AS VARIANT
    DIM puArgErr AS DWORD
    ppenum = 0
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER
    FUNCTION = TB_Collection_IDispatch_Invoke (pthis, -4,
IID_NULL, 0, 1 OR 2, uDispParams, vResult, BYVAL 0, puArgErr)
    ppenum = VARIANT#(vResult)
    IF ISTRUE ppenum THEN TB_Collection_IUnknown_AddRef ppenum
' increment the reference counter
END FUNCTION

' Returns the number of objects of the collection.
' Provides the same functionality as the Count property of the
object.

FUNCTION TB_Collection_Count (BYVAL pthis AS DWORD) AS LONG

    LOCAL HRESULT AS LONG               ' // COM result code
    LOCAL IID_IEnumVariant AS GUID      ' // GUID of the
IEnumVARIANT interface
    LOCAL pEnum AS DWORD                ' // Address of a
pointer to the collection
    LOCAL pIEnumVARIANT AS DWORD        ' // Address of a
pointer to the IEnumVARIANT interface
    LOCAL nCount AS LONG                ' // Number of elements
in the collection
    LOCAL celtFetched AS DWORD          ' // Number of elements
fetched
    LOCAL dwArray AS DWORD              ' // Pointer to the
first element in the array
    LOCAL vVar AS VARIANT               ' // General purpose
variant

    IID_IEnumVARIANT = GUID$("{00020404-0000-0000-c000-
000000000046}")

    ' Check for null pointer
    IF ISFALSE pthis THEN EXIT FUNCTION

    ' Get a reference to the Enumerator object
    HRESULT = TB_Collection_NewEnum (pthis, pEnum)
    IF ISTRUE HRESULT OR ISFALSE pEnum THEN EXIT FUNCTION

    ' Get a pointer to the IEnumVARIANT interface.
    HRESULT = TB_Collection_IUnknown_QueryInterface (pEnum,
IID_IEnumVARIANT, pIEnumVARIANT)
    IF ISTRUE HRESULT OR ISFALSE pIEnumVARIANT THEN
       TB_Collection_IUnknown_Release pEnum
       EXIT FUNCTION
    END IF

    ' Release the collection's enumerator interface.
    ' Note: If pEnum and pIEnumVARIANT are the same, this will
decrease the reference count;
```

```
    ' otherwise, it will release the enumerator interface.
    TB_Collection_IUnknown_Release pEnum

    ' Position the enumerator at the beginning of the list
    HRESULT = TB_Collection_IEnumVARIANT_Reset(pIEnumVARIANT)
    IF ISTRUE HRESULT THEN
       TB_Collection_IUnknown_Release pIEnumVARIANT
       EXIT FUNCTION
    END IF

    ' Parses the collection
    DO
       ' Fetch an element of the collection
       HRESULT = TB_Collection_IEnumVARIANT_Next (pIEnumVARIANT,
1, BYVAL VARPTR(vVar), celtFetched)
       IF ISTRUE HRESULT OR celtFetched < 1 THEN EXIT DO
       nCount = nCount + 1
    LOOP

    ' Release the interface
    TB_Collection_IUnknown_Release pIEnumVARIANT

    ' Return the number of objects retrieved
    FUNCTION = nCount

  END FUNCTION

' TB_EnumCollection
' - Helper function to enumerate collectios. Returns an array of
pointers to objects.
'   This provides similar functionality to the Visual Basic's
ForEach function.
'   We need it to enumerate collections than expect a key
instead of an index.
'   It is also much faster to enumerate an entire collection
using this method that using
'   the Item property, that should be reserved to get a single
object of the collection.
' Parameters:
'   pthis   = Pointer to the interface whose collection we want
to enumerate.
'   vArray  = Dimensioned array of variants.
' Return Value:
'   An HRESULT (&H80004003 = %E_POINTER) or error 461 (array not
dimensioned)
' Example:
'
' The code below demonstrates the use of the enumerator with the
FileSystemObject
' using PB Automation.
'
'  #DIM ALL
'  #DEBUG ERROR ON
'  #INCLUDE "win32api.inc"
'  #INCLUDE "TB_ENUM.inc"
'
```

```
'  FUNCTION PBMAIN
'
'    ' Create an instance of the object.
'    LOCAL oFso AS DISPATCH
'    SET oFso = NEW DISPATCH IN "Scripting.FileSystemObject"
'    IF ISFALSE ISOBJECT(oFso) THEN EXIT FUNCTION
'
'    ' Get a reference to the Folder object
'    LOCAL oFolder AS DISPATCH
'    LOCAL vFolder AS VARIANT
'    LOCAL vPath AS VARIANT
'    vPath = "c:\pbwin70\bin\"
'
'    OBJECT CALL oFso.GetFolder(vPath) TO vFolder
'    SET oFolder = vFolder
'
'    ' Get a reference to the Files collection
'    LOCAL oFiles AS DISPATCH
'    LOCAL vFiles AS VARIANT
'    OBJECT GET oFolder.Files TO vFiles
'    SET oFiles = vFiles
'
'    ' Release the Folder interface
'    SET oFolder = NOTHING
'
'    ' Get the number of files
'    LOCAL vFilesCount AS VARIANT
'    LOCAL nCount AS LONG
'    OBJECT GET oFiles.Count TO vFilesCount
'    nCount = VARIANT#(vFilesCount)
'
'    ' Enumerate the Files collection
'    LOCAL oItem AS DISPATCH
'    LOCAL i AS LONG
'    LOCAL vName AS VARIANT
'    LOCAL vRes AS VARIANT
'
'    DIM vArray (1 TO nCount) AS VARIANT           ' // DIM an
array of variants
'    TB_EnumCollection(OBJPTR(oFiles), vArray())  ' // Enumerate
the collection
'
'    FOR i = LBOUND(vArray) TO UBOUND(vArray)
'       SET oItem = vArray(i)                      ' // Assign
the interface reference
'       IF OBJRESULT THEN EXIT FOR                 ' // Exit on
failure
'       vName = EMPTY                              ' // Empty
the variant
'       OBJECT GET oItem.Name TO vName             ' // Get the
name of the file
'       PRINT VARIANT$(vName)                      ' // Show it
'       SET oItem = NOTHING                        ' // Release
the interface
'    NEXT
'
```

```
'    SET oFiles = NOTHING  ' Release the Files interface
'    SET oFso = NOTHING    ' Release the FileSystemObject object
'
'    WAITKEY$
'
'  END FUNCTION

FUNCTION TB_EnumCollection (BYVAL pthis AS DWORD, vArray() AS
VARIANT) AS LONG

    LOCAL HRESULT AS LONG               ' // COM result code
    LOCAL IID_IEnumVariant AS GUID      ' // GUID of the
IEnumVARIANT interface
    LOCAL pEnum AS DWORD                ' // Address of a
pointer to the collection
    LOCAL pIEnumVARIANT AS DWORD        ' // Address of a
pointer to the IEnumVARIANT interface
    LOCAL nCount AS LONG                ' // Number of elements
to fetch
    LOCAL celtFetched AS DWORD          ' // Number of elements
fetched
    LOCAL dwArray AS DWORD              ' // Pointer to the
first element in the array
    LOCAL vVar AS VARIANT               ' // General purpose
variant

    IID_IEnumVARIANT = GUID$("{00020404-0000-0000-c000-
000000000046}")

    ' Check for null pointer
    IF ISFALSE pthis THEN FUNCTION = &H80004003 : EXIT FUNCTION
' %E_POINTER

    ' Number of elements in the array
    nCount = ARRAYATTR(vArray(), 4)
    IF nCount = 0 THEN FUNCTION = 461 : EXIT FUNCTION   ' Array
not dimensioned

    ' Get a reference to the Enumerator object
    HRESULT = TB_Collection_NewEnum (pthis, pEnum)
    IF ISTRUE HRESULT THEN FUNCTION = HRESULT : EXIT FUNCTION
    IF ISFALSE pEnum THEN FUNCTION = &H80004003 : EXIT FUNCTION

    ' Get a pointer to the IEnumVARIANT interface.
    HRESULT = TB_Collection_IUnknown_QueryInterface (pEnum,
IID_IEnumVARIANT, pIEnumVARIANT)
    IF ISTRUE HRESULT THEN
       FUNCTION = HRESULT
       TB_Collection_IUnknown_Release pEnum
       EXIT FUNCTION
    END IF

    IF ISFALSE pIEnumVARIANT THEN
       FUNCTION = &H80004003
       TB_Collection_IUnknown_Release pEnum
       EXIT FUNCTION
```

```
    END IF

    ' Release the collection's enumerator interface.
    ' Note: If pEnum and pIEnumVARIANT are the same, this will
decrease the reference count;
    ' otherwise, it will release the enumerator interface.
    TB_Collection_IUnknown_Release pEnum

    ' Position the enumerator at the beginning of the list of
elements
    HRESULT = TB_Collection_IEnumVARIANT_Reset (pIEnumVARIANT)

    IF ISTRUE HRESULT THEN
       FUNCTION = HRESULT
       TB_Collection_IUnknown_Release pIEnumVARIANT
       EXIT FUNCTION
    END IF

    ' Fetch nCount elements of the collection
    dwArray = VARPTR(vArray(LBOUND(vArray)))
    HRESULT = TB_Collection_IEnumVARIANT_Next (pIEnumVARIANT,
nCount, dwArray, celtFetched)
    FUNCTION = HRESULT

    ' Release the interface
    TB_Collection_IUnknown_Release pIEnumVARIANT

END FUNCTION
```

Note: Because of the call to the Invoke method, teh above enumerator can only be used with collections that implement a dual or dispatch interface, that is the usual case. Collections found in IUnknown-only interfaces must be enumerated using the methods provided by the collection's interface, that include a function to return a pointer to the enumerator and the four standard methods Next, Skip, Reset and Clone. The code below shows hot to enumerate the WorksItems collection of the Task Scheduler:

```
' This example enumerates all the tasks in the Scheduled Tasks
folder of the local computer.

#COMPILE EXE
#DIM ALL
#INCLUDE "Win32Api.inc"

%CLSCTX_INPROC_SERVER = &H1

' Release method
' 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.
```

```
FUNCTION IUnknown_Release (BYVAL pthis AS DWORD PTR) AS DWORD
    LOCAL DWRESULT AS DWORD
    CALL DWORD @@pthis[2] USING IUnknown_Release(pthis) TO
DWRESULT
    FUNCTION = DWRESULT
END FUNCTION

' Enum method
' The Enum method retrieves a pointer to an OLE enumerator
object that enumerates the tasks in
' the current task folder.

FUNCTION ITaskScheduler_Enum (BYVAL pthis AS DWORD PTR, BYREF
ppEnumWorkItems AS DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[5] USING ITaskScheduler_Enum(pthis,
ppEnumWorkItems) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

' Next method
' The Next method retrieves the next specified number of tasks
in the enumeration sequence. If
' there are fewer than the requested number of tasks left in the
sequence, all the remaining
' elements are retrieved.

FUNCTION IEnumWorkItems_Next (BYVAL pthis AS DWORD PTR, BYVAL
celt AS DWORD, BYREF rgpwszNames AS DWORD, BYREF pceltFetched AS
DWORD) AS LONG
    LOCAL HRESULT AS LONG
    CALL DWORD @@pthis[3] USING IEnumWorkItems_Next(pthis, celt,
rgpwszNames, pceltFetched) TO HRESULT
    FUNCTION = HRESULT
END FUNCTION

FUNCTION PBMAIN

    LOCAL hr AS LONG
    LOCAL CLSID_CTaskScheduler AS GUID
    LOCAL IID_ITaskScheduler AS GUID
    LOCAL pITS AS DWORD
    LOCAL pIEnum AS DWORD
    LOCAL TASKS_TO_RETRIEVE AS DWORD
    LOCAL dwFetchedTasks AS DWORD
    LOCAL rgpwszNames AS DWORD PTR
    LOCAL buffer AS STRING
    LOCAL bstrlen AS LONG
    LOCAL i AS LONG

    ' Create a Task Scheduler object
    CLSID_CTaskScheduler = GUID$("{148BD52A-A2AB-11CE-B11F-
00AA00530503}")
    IID_ITaskScheduler = GUID$("{148BD527-A2AB-11CE-B11F-
00AA00530503}")
```

```
    hr = CoCreateInstance(CLSID_CTaskScheduler, BYVAL %NULL,
%CLSCTX_INPROC_SERVER, IID_ITaskScheduler, pITS)
    IF ISTRUE hr OR ISFALSE pITS THEN EXIT FUNCTION

    ' Call ITaskScheduler_Enum to get an enumeration object
    hr = ITaskScheduler_Enum(pITS, pIEnum)
    ' Release the Task Scheduler interface
    IUnknown_Release pITS
    ' Terminate if ITaskScheduler_Enum has failed
    IF ISTRUE hr OR ISFALSE pIEnum THEN EXIT FUNCTION

    ' Call IEnumWorkItems_Next to retrieve tasks. Note that
    ' this example tries to retrieve five tasks for each call

    TASKS_TO_RETRIEVE = 5

    DO
       ' Retrieve the tasks
       hr = IEnumWorkItems_Next(pIEnum, TASKS_TO_RETRIEVE,
rgpwszNames, dwFetchedTasks)
       IF ISFALSE dwFetchedTasks THEN EXIT DO
       IF ISTRUE rgpwszNames THEN
          FOR i = 0 TO dwFetchedTasks - 1
             ' Extract the name (unicode) and show it
             bstrlen = lstrlenW(BYVAL @rgpwszNames[i])
             IF ISTRUE bstrlen THEN
                buffer = PEEK$(@rgpwszNames[i], bstrlen * 2)
                MSGBOX ACODE$(buffer)
             END IF
             ' Free the task name
             CoTaskMemFree @rgpwszNames[i]
          NEXT
       END IF
    LOOP

    ' Release the array
    IF ISTRUE rgpwszNames THEN CoTaskMemFree rgpwszNames

    ' Release the collection
    IF ISTRUE pIEnum THEN IUnknown_Release pIEnum

END FUNCTION
```