Designing COM Interfaces

Charlie Kindel
Program Manager, Windows NT

October 20, 1995

Abstract

This paper discusses the design of Microsoft® Component Object Model (COM) interfaces. Readers are assumed to have a good understanding of COM as well as a basic knowledge of the features of OLE. However, because this paper discusses COM from a different angle than other works, even expert COM developers can learn something from it.

Introduction

The Microsoft® OLE specification defines a large number of Component Object Model (COM) interfaces. Some of these interfaces are tied very closely to a feature (such as **IOleInPlaceActiveObject**), but many are general-purpose interfaces applicable to large problem domains (such as **IDataObject**). In the process of developing components that need to interoperate with other binary components, many developers have found the set of interfaces defined by OLE suitable for their needs. This is particularly true for independent software vendors (ISVs) focusing on allowing their applications to integrate in more traditional OLE scenarios such as OLE documents or controls. However, many vendors are discovering the power of using COM *within* their own products for tasks such as managing third-party add-in modules or sharing components between the applications that comprise a product line. In addition, many developers are using COM to design and implement enterprise-wide distributed computing solutions. Thus, the need for ISVs to design their own COM interfaces is increasing.

A big part of the "philosophy" of COM is a clear separation between interface definition and implementation. There are many articles (and books!) available that explain everything there is to know about *implementing* COM interfaces. This paper focuses on the rules, procedures, and philosophy of *defining* COM interfaces.

The paper is separated into four parts:

- Part 1 discusses the COM rules for interface design.
- Part 2 discusses the elements that make up an interface design.
- Part 3 provides design patterns for designing interfaces.
- Part 4 provides a tutorial for using the Microsoft Interface Definition Language (MIDL) compiler to generate the proxy and stub objects for an interface.

Much of the information presented here can be found in the various Microsoft OLE and remote procedure call (RPC) documentation. However, this paper presents the relevant information as a cohesive whole. In addition, there are tips and pointers in this paper that are anecdotal in nature, generated through interviews and conversations with developers and designers who have real-world experience designing and implementing COM interfaces.

Readers interested in gaining a better understanding of what the Component Object Model is, as well as the motivations behind its design and philosophy, should read the first two chapters of the *Component Object Model Specification* (MSDN Library, Specifications). Chapter 1 is a brief introduction, and Chapter 2 provides a thorough overview.

Throughout this paper, interface definition language (IDL) syntax is used to describe interfaces. Readers are encouraged to read Chapter 12, "Interface Definition Language," in the COM specification, and the *RPC Programmer's Guide* in the Platform SDK.

Part 1: Interface Rules

This section lists the rules Microsoft provides as part of the COM specification that are specific to the design of interfaces. There are other rules for using interfaces and implementing COM objects; please refer to my article "The Rules of the Component Object Model" for a complete discussion of all these rules.

"Remotable" vs. In-Process-Only Interfaces

Note that the flexibility of COM allows an interface designer extreme freedom. For example, interface designers can specify any data type as an argument for a method. One implication of this is that interfaces can be designed that are physically impossible to "remote" (call cross-process). A concrete example would be an interface that takes an argument that represents something that only makes sense within the context of the current process, like a Win32 graphics device interface (GDI) device context.

In general, all COM interfaces should be designed so that they support distributed processing. This paper discusses the design of interfaces with this general rule in mind. However, there are clearly circumstances where an interface will only be used in an in-process case, so we also try to point out those rules that can be bent for the "in-process-only" case.

Interfaces Derive from IUnknown

All COM interfaces must derive directly or indirectly from the **IUnknown** interface. In other words, any interface implemented on a COM object must have as its first three methods **QueryInterface**, **AddRef**, and **Release**, in that order.

Thus, when describing any COM interface in IDL, you will use syntax such as this:

```
import "unknwn.idl";

[ object, uuid(4411B7FE-EE28-11ce-9054-080036F12502) ]
interface ISome : IUnknown
{
    HRESULT SomeMethod(void);
};

[ object, uuid(4411B7FD-EE28-11ce-9054-080036F12502) ]
interface ISomeOther : ISome
{
    HRESULT SomeOtherethod([in]long l);
};
```

The interface **ISome** contains four methods: **QueryInterface**, **AddRef**, **Release**, and **SomeMethod**. The interface **ISomeOther** contains five methods: all of the methods in **ISome**, plus **SomeOtherMethod**.

Interfaces Must Have a Unique Identifier

Remember that the "real" name of an interface is a 128-bit globally unique identifier (GUID), not its human-readable name. Thus, for each newly defined interface, a new *interface identifier*, or IID**,** must be generated. You can either use UUIDGEN.EXE or GUIDGEN.EXE to generate a new GUID that will be your interface's IID. UUIDGEN is a console application that is part of Microsoft RPC, and GUIDGEN is a Microsoft Windows® application that is included with Microsoft Visual C++®; however, they both are functionally the same, eventually calling the distributed computing environment (DCE) RPC run-time application programming interface (API) **UuidCreate**. (UUIDGEN calls **UuidCreate** directly; GUIDGEN calls the COM API **CoCreateGuid**, which is simply a wrapper around **UuidCreate**.)

Note in the IDL sample above that each interface has its own unique IID. By convention, symbolic constants used to identify a specific IID are of the form IID_<interface name>. Thus, for our example above, **IID_ISome** would represent the IID value 4411B7FE-EE28-11ce-9054-080036F12502.

Interfaces Are Immutable

After it is published, the interface contract associated with a particular IID can never change.

Functions Should Return HRESULTs

All methods in your interface should return values of type **HRESULT**. Note that this does not apply to **IUnknown::AddRef** and **IUnknown::Release,** which are exceptions. ("**HRESULT**" implies "handle to a result". This is an historical oddity: **HRESULT** is synonymous with **SCODE**. So when you read "**HRESULT**", just think "status code".)

While it is possible for COM interface functions to return types other than **HRESULT**, the interfaces you design should not do so. The reason is that the COM remoting infrastructure (see Chapter 7 of the COM specification) needs to return RPC errors to the caller. If you define methods with return types other than **HRESULT**, COM has no way to tell callers of your methods that you have crashed (**RPC_E_SERVERDIED**) or that the network has gone down (**RPC_E_COMM_FAILURE**). **AddRef** and **Release** are specifically defined such that they *cannot* return errors, which explains why they are excepted from this rule.

Strictly speaking, for interfaces that are not intended to ever be remoted, this rule can be ignored. However, even for interfaces that are intended to be implemented by in-process objects only and never passed across a process boundary, we recommend that you follow the rule for programming model consistency.

String Parameters Should Be Unicode

All strings passed through all COM interfaces (and, on Microsoft platforms, all COM APIs) are Unicode™ strings. There simply is no other effective way to implement interoperable objects in the face of an architecture that provides call-location transparency, and doesn't in all cases intervene system-provided code between client and server.

Part 2: Elements of an Interface

Like any academic field, the field of object-oriented programming (OOP) analysis and design has many areas of debate. However, the concept of a separation of interface from implementation is well agreed upon (can you say encapsulation?). One OOP scholar says that we should concentrate upon the outside view of an object, and he calls this *contract programming* (B. Meyer, *Object-Oriented Software Construction*, 1988). Contract programming is central to the philosophy of COM. A COM interface defines a contract between the implementor and the user that physically prevents the user from accessing any of the details of the implementation.

The designer of an interface is responsible for documenting all of the information that describes the contract represented by the interface. The elements that must be included in a COM interface contract are listed below, with a more complete discussion following.

- Interface ID
  - Human-readable name
- Interface signature. Each method has a signature that includes the following:
  - Method name
  - Return value type
  - Parameter names
  - Parameter order and type
- Interface semantics
- Marshalling buffer format

Below we describe in detail all of the elements that make up an interface definition.

Interface Identifiers

Each interface must have a GUID that serves as its programmatic name. It is this interface ID (IID) that uniquely identifies the contract defined by the interface. After an interface design with a particular IID is published, the specifics of all the other

elements (described in detail below) that make up that interface *cannot change . . . ever.* (By *published*, we mean implemented in a binary component and "released" to another party to use.)

Newcomers to COM may find the previous statement to be too strong. However, the immutability of an interface contract is fundamental to the power and robustness of COM. If an interface design turns out to be insufficient or faulty in some way after it is published, a completely new contract must be drawn up—that is, a new IID must be generated.

Interfaces should have *human-readable* names; that is, they should be given names that indicate the services the interface exposes. By convention, interface names begin with the capital letter "I". Assigning human-readable names to interfaces, methods, and method parameters is technically optional; however, including them makes the life of human programmers significantly more pleasant.

Interface Signature

The *interface signature*, also sometimes called the *interface syntax*, provides the user of an interface with enough information to push parameters onto the stack and determine which offset into the vtable to use as the method address. Specifically, the interface signature defines the following:

- The number and order of methods in the interface;
- The number, order, and type of each parameter for each of the methods; and
- The return value type for each of the methods.

For parameters, the type includes whether the parameter is an *in*, an *out*, or an *in-out* parameter. The interface signature also includes type definitions (for example, structures) used in the interface and calling convention (**cdecl**, **Pascal**, **__stdcall**, and so on.)

The values of any constants associated with the interface (including failure and success codes) are also part of the signature, but the *meaning* of those constants is part of the semantics (see "Defining **HRESULT** values" below). Strictly speaking, each method in an interface could legally have a different calling convention; however, calling conventions are very platform-specific, and, in terms of binary interoperability, a calling convention is important only on a per-platform basis. Thus it is highly recommended that interfaces specify that the default calling convention for the platform be used. Do this by simply not specifying a calling convention.

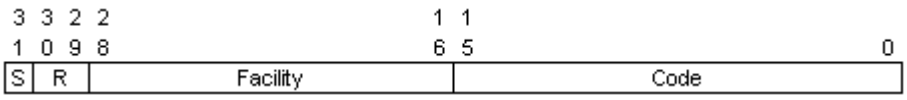
Defining HRESULT values

The COM specification provides rules regarding the definition of new status codes (**HRESULTs**). Section 3.4.1. of the COM specification explains all of this in detail, but we give a short overview here.

COM interface methods and COM Library API functions use a specific convention for error codes in order to pass back to the caller both a useful return value and an indication of status or error information. For example, it is highly useful for a function to return a Boolean result (TRUE or FALSE), as well as to indicate failure or success. (Whereas an error code indicates that the function failed completely, returning TRUE and FALSE means that the function executed successfully, and TRUE or FALSE is also the answer.)

The **HRESULT** type (a 32-bit integer) is the medium through which these status codes are passed.

An **HRESULT** has an internal structure comprised of four fields with the following format (numbers indicate bit positions):



S:	(1 bit) Severity field:
	0 *Success*. The function was successful; it behaved according to its proscribed semantics.

	1 *Error*. The function failed due to an error condition.
R:	(2 bits) Reserved for future use; must be set to zero by present programs generating **HRESULTs**; present code should not take action that relies on any particular bits being set or cleared this field.
Facility:	(13 bits) Indicates which group of status codes this belongs to. New facilities must be allocated by Microsoft because they need to be universally unique. However, the need for new facility codes is very small. In most cases, you can and should use FACILITY_ITF.
Code:	(16 bits) Describes what actually took place, error or otherwise.

All facility codes, except FACILITY_ITF, are reserved for COM-defined failure and success codes. In other words, if you need to define failure or success codes that are specific to the interfaces you are designing, you must set their facility to FACILITY_ITF.

Status codes in facility FACILITY_ITF are defined solely by the creator of the interface. That is, in order to avoid conflicting error codes, a human being needs to coordinate the assignment of codes in this facility, and it is he or she who defines the interface that does the coordination.

Likewise, it is possible (though not required) for designers of suites of interfaces to coordinate the error codes across the interfaces in that suite so as to avoid duplication. The designers of the OLE Documents interface suite, for example, ensured such lack of duplication.

All the COM-defined FACILITY_ITF codes have a code value that lies in the region 0x0000–0x01FF. Thus, while it is indeed legal for the definer of a new interface to make use of any codes in FACILITY_ITF, we strongly recommend that you use code values only in the range 0x0200–0xFFFF, so that you reduce the possibility of accidental confusion with any COM-defined errors. We also strongly recommend that you consider defining as legal that most if not all of your functions can return the appropriate status codes defined by COM in facilities other than FACILITY_ITF. For example, E_UNEXPECTED is an error code that you will probably want to make universally legal.

Interface Semantics

Just looking at the names of methods and method parameters can often give the caller clues about what function the methods perform when called, but they usually can't glean enough information in this manner. The *semantics* of the interface are the parts of the contract that the signature alone cannot describe. *Interface semantics* include descriptions of the behavior of each method, the context and order in which the method can or should be called, the failure codes specific to the method, and the possible success codes. (This use of the word *semantics* is yet another bastardization of the English language by us computer folk. It wasn't my idea, but I apologize anyway.)

You may find it useful to think of interface semantics in terms of invariants. An *invariant* is some condition that must be true. For interfaces, there are both pre- and post-condition invariants. An example of a pre-condition invariant would be "the *pUnk* parameter must never be NULL when calling method **Some**." An example of a post-condition invariant would be "on successful return ***ppUnk* will point to the newly created object." If the implementor violates an interface invariant, then they have broken the contract. Therefore, when defining an interface, be sure to clearly document all invariants associated with it.

Most C developers are familiar with documenting how to call some function they've written. This documentation usually consists of a little header block immediately preceding the function, such as the following:

```
// ERRCODE WakeUp(HROBOT hRobot, WCHAR* pwcsMsg)
// Function Description:
//   WakeUp causes the robot to transition from sleeping
//   to ready state, announcing its readiness by displaying
//   (or speaking) a message.
```

```
// Parameters:
//   hRobot   Handle to the robot instance to be woken up. Must not
//            be NULL.
//   pwcsMsg  Pointer to a NULL-terminated Unicode string containing
//            the message the robot is to display or pronounce upon
//            completing the transition to ready state. pwcsMsg may
//            be NULL, in which case the robot will use a default
//            string (the default string can be changed with the
//            SetWakeUpString function).
// Return value:
//   This function may return the following ERRCODEs:
//     ERR_OK
//            Transition to ready state was successful.
//     ERR_ALREADY_READY
//            The robot was already in the ready state. No message
//            was displayed.
//     ERR_HW_FAILURE
//            A hardware error prevented the robot from transitioning
//            to the ready state.
//     ERR_INVALIDARG
//            One or more of the parameters is invalid.
```

The first line of this sample is the definition of the *signature* of the **WakeUp** function, which is part of a (fictional) API for controlling robots. The remaining information documents the function's *semantics*. The semantics of the entire robot API is simply the sum of all of the individual-function semantics.

To further emphasize the idea of pre- and post-condition invariants, consider the documentation of the *hRobot* and *pwcsMsg* parameters in this function. The condition that *hRobot* must not be NULL is clearly a pre-condition invariant. The condition that a non-NULL *pwcsMsg* will be displayed by the robot indicates a post-condition invariant: if calling **WakeUp** with a non-NULL *pwcsMsg* results in a successful return (ERR_OK or ERR_ALREADY_RUNNING) and the robot fails to display the message, then the robot has failed to carry out its side of the contract.

Documenting the semantics of a COM interface is really no different than documenting the semantics of a set of related C APIs. You should document the semantics of each method that makes up the interface, just as you would any other API function, and then follow some simple rules to bring them all together into one interface definition.

The following items are typically included in the semantic description of a method:

- Pre-conditions
- Post-conditions
- Failure codes
- Success codes

If we were to design our robot API to be COM-based, the semantic description for the **IRobotBrain::WakeUp** method might look something like this:

```
HRESULT WakeUp([in] WCHAR* pwcsMsg)
```

Causes the robot to transition from the sleeping state to the ready state, announcing its readiness by displaying (or speaking) a message.

Argument	Type	Description
pwcsMsg	**WCHAR***	Pointer to a NULL-terminated Unicode string containing the message the robot is to display or pronounce upon completing the transition to ready state. *pwcsMsg* may be NULL, in which case the robot will use a default string (which can be changed with the **IRobotBrain::SetWakeUpString** function).

Return Value	Meaning
S_OK	The robot successfully made the transition from sleeping to ready state.
ROBOT_S_ALREADY_READY	The robot was already in the ready state. No message was displayed.
ROBOT_E_HW_FAILURE	A hardware error prevented the robot from making the transition to ready state.
E_UNEXPECTED	An unspecified error occurred.
Other errors	Other failures are possible and should be handled generically.

Specifying the function that a particular method performs is usually simple. It is important to include in the description of the method the state changes that occur because of its execution (the post-condition invariants).

Because of COM's location transparency, the complete set of failure codes that a particular method can return is undefined. However, you can define **HRESULTs** that are specific to a method or interface (such as ROBOT_E_HW_FAILURE in our example), and you can specify under what conditions a predefined failure code (for example, E_UNEXPECTED) can be returned. (*Location transparency* is the feature of COM that enables COM clients to access objects that are running in-process, on the same machine but out-of-process, or on a different machine in the same way.)

You also define the success codes that a method can return, and callers can assume that the success codes you specify in the contract are the only ones that will be returned. Many interface methods return only a single success code, S_OK. In our example, we indicate that an additional success code that we define, ROBOT_S_ALREADY_AWAKE, can be returned.

Marshalled Format

When a COM method call is made across a process boundary (or thread, in the case of the apartment threading model), the parameters to the method must be marshalled into a buffer to be sent "over the wire." (We use the term *wire*, here, to represent even the "logical" wire that exists between two processes on the same machine.) On the other side, this buffer must be unmarshalled into the correct stack frame, and the actual implementation must be called. Upon return, the process reverses: The return value and any out parameters are marshalled and sent on the wire, and then they are unmarshalled before returning to the caller. The interface designer is responsible for specifying the format of the *marshalling buffer*. (Generally, the designer of the interface is also responsible for providing the code that actually does the marshalling; however, because the interface definition includes the format of the marshalled buffer, anyone should be able to faithfully produce a proxy or stub for the interface on any platform.)

Only rarely do you care about defining the format of the marshalling buffer, because there are many predefined RPC wire representations. One such representation is network data representation (NDR), which is the format used by the Microsoft IDL (MIDL) compiler. Therefore, many interface designers define their interface using MIDL and specify that the buffer format is NDR. It is extremely important to note that IDL is simply one tool that COM interface designers can use. It is a popular tool for the task as well, but COM in no way relies on IDL or NDR to achieve its binary component interoperability. Also note that, for interfaces that are designed to be implemented by in-process objects only, the requirement of specifying the format of the marshalling buffer is waived.

The Whole Interface

Just as each method in an interface requires a description of its function and context, so does the interface as a whole.

It should be clear to the reader of an interface definition what component implements the interface and which component uses it. You'll notice that the description for **IRobotBrain::WakeUp** is written with the interface *user* in mind. When designing an interface and describing its semantics, think about who is more likely to read your documentation: implementors of the interface or users? Then tailor your wording to the appropriate audience.

The most recent generation of the Platform SDK documentation includes sections describing "when to implement" and "when to use" a particular OLE interface. These sections clear up a lot of confusion about the intent of the interface contract. You should consider copying this style when documenting your interfaces.

Interface Suites

Up to this point we have been discussing the design of interfaces as though each individually designed interface stands alone. In practice, an interface design actually includes the design of several cooperating or related interfaces. A group of cooperating or related interfaces is called an *interface suite*. Examples of interface suites defined by Microsoft include:

Interface Suite Name	Description	Interfaces
OLE Documents	Enables compound document functionality, including linking, embedding, and visual editing.	Interfaces whose names begin with **IOle**...
Persistent Objects	Enables objects to save their persistent state to an opaque medium controlled by the client.	**IPersist**, **IPersistStream**, **IPersistStreamInit**, **IPersistStorage**, **IPersistFile**
Uniform Data Transfer	Rich-data transfer mechanisms.	**IDataObject**, **IAdviseSink**, **IAdviseSinkHolder**
Connectable Objects	Enables "outgoing" interfaces for events and notifications, as well as networks (for example, trees) of objects.	**IConnectionPoint**, **IConnectionPointContainer**, **IEnumConnectionPoints**

When you document your interface definition, you should state the relationships between the various interfaces that comprise your interface suite. In addition, it is likely that your suite will actually rely on interfaces from other suites (OLE Documents, for example, relies on interfaces from Uniform Data Transfer as well as those from Persistent Objects). The readers of your interface definition should be able to easily understand these relationships.

Part 3: Interface Design Issues

Parts 1 and 2 of this paper described interface design issues that are related to hard and fast rules. Part 1 specifically listed several "laws" specified by Microsoft in the COM specification. Part 2 specified the elements of an interface design. However, if simply knowing a few rules were all there was to designing COM interfaces, COM probably wouldn't be very useful. COM allows incredible leeway in how interfaces can be designed, while still providing a binary standard. However, this flexibility places a burden on the interface designer to design carefully and thoughtfully.

Therefore, this section advises you on *how* to define interfaces rather than *what* the contents of an interface definition are. We do not claim that any of the advice given here is the only or best advice there is. The fact that we admit that it is advice should be telling to the reader.

Naming

Programmatically, the only name that really matters for a given interface is the GUID that is its IID. However, humans need more descriptive names in order to quickly identify a particular interface as meeting a need. Thus, you need to assign your interfaces human-readable names. There are no hard and fast rules for you to use, but the following general guidelines may be useful:

- Make the name instantly and intuitively convey the purpose of the interface. If the interface represents a service, then use the name of that service as part of the interface name. For example, **IOleInPlaceActiveObject** is a good name, but **ISomeInterface** is not.
- Strive to make the human-readable name as unique as possible. If the interface has broad applicability (that is, you believe it to be general-purpose), you will be tempted to give it a very generic name. Over time, it is likely that some other developer will design an interface with similar functionality and the same human-readable name. The recommended convention is to prefix the service name with an indication of which interface suite the interface belongs to. For example, **IOleObject** belongs to the OLE Documents interface suite. Another example is **IMAPIPropertySet**.
- If the interface is an enumerator (see the section below on COM enumerators), use the naming convention established for COM enumerators (that is, prefix the name with **IEnum**).
- When creating a new version of an existing interface, use a version number as a suffix rather than something like "**Ex**". For example, **IRobotBrain2** is good, but **IRobotBrainEx** is bad. You may someday need to introduce a new version of **IRobotBrainEx**, and no one wants to see an interface named **IRobotBrainExEx**.

Factoring

One of the biggest issues to be considered when designing interfaces is factoring. *Factoring* is the process by which you decide how many interfaces to design, how many methods each of the interfaces have, and how many parameters each of the methods has. An entire book could be written on strategies for factoring interfaces, and there is much literature available on the topic of object-oriented analysis and design that is applicable. However, there are some basic rules you can use as you design your interfaces. These rules are described in the following sections.

Number of methods per interface

Experience has shown that interfaces with fewer methods are better. Interfaces with many methods that are intended to be implemented by a large number of objects usually end up having most of the methods return E_NOTIMPL.

Fewer methods, however, means more interfaces. The greater the number of interfaces, the greater the number of times a client might be forced to call **QueryInterface** just to execute a simple task.

The general rule is if two sets of functions are independent—that is, you expect either to be implemented without the other—the sets of functions should be contained in different interfaces. In most cases, if you are tempted to have a "capability flag" to indicate whether some functions are implemented, you should separate interfaces and take advantage of **QueryInterface** instead.

Also, try to eliminate options no one will want to use or implement. Often, interface designers try to think up every conceivable use for their interface and thus add additional methods to satisfy these "potential" users. Do not fall into this trap. Instead, focus on your primary users and design the interface so that it fits their needs. If a customer needs additional, special functionality, you can provide that functionality in another interface.

Number of parameters per method

When factoring your design, think about "round trips." Each call to an interface method involves at least one "round trip," potentially across a process or machine boundary. Therefore, it is "cheaper" to send everything needed to execute a call with one method than to have to call two methods with half as many parameters. However, it is sometimes possible to reduce the amount of data marshalled by doing just the opposite: have one "setup" method and then let users call the various "worker" methods without having to supply the "setup" information each time.

Also, try to limit the number of parameters a method contains. Having to call a method that takes more than five or six parameters is bothersome to many programmers (and you may start reaching the bounds of what the programmer's compiler can handle).

Contract Strictness

The contract specified by the interface definition may be either rigid or flexible. A flexible contract lets implementors return E_NOTIMPL for one or more methods. A rigid contract requires that all methods be fully implemented. Experience shows that, in general, rigid contracts are better than flexible ones. Consider the following factors when deciding how strict your contract should be:

- Is this interface to be implemented by many objects? If the answer is no, because only a few implementations will ever exist (**IStorage** is a good example of this), the contract almost certainly should be very strict.
- Are there other mechanisms you can use to express the reduced functionality? Returning E_NOTIMPL is often the first clue a user of an interface has that he or she is dealing with a "reduced functionality" implementation. The user may have already executed quite a bit of code by the time the user has made the specific call, and backing out may be difficult. Factoring the interface into two or more additional interfaces is often the best way to address this problem.

Any time you decide that a method can return E_NOTIMPL, you should clearly say so when you describe the semantics of the interface. It is equally important to both implementors and users of interfaces that an interface can return E_NOTIMPL.

Success Codes

The concept of being able to return success codes other than "TRUE" is powerful, but also confusing to novices.

Use of Unions

If you have more than two unions in the method syntax, something is wrong with your design. Unions are slow, confusing, and generally are not good data types for interprocess communications (IPC). Note that unions should not be used as a base type.

Naturally Align Structures

Obvious point, but well worth noting. The default for MIDL is 8-byte packing, and is generally efficient on all CPU architectures.

By-Value Parameters

Avoid passing structures and unions by-value. Passing by pointer is more natural for C programmers, and when going cross-process a copy is made anyway.

Passing Interface Pointers

When passing interface pointers as either in or out parameters, do not use *void*** as the type. Use *IUnknown*** instead. The *void*** arguments used in **QueryInterface** are an exception because MIDL inherently understands the semantics of the function.

```
interface ISome : IUnknown
{
    ...
    HRESULT CreateSubObject([in]REFIID riid, [out, iid_is(riid)]IUnknown**
                            ppUnk);
    ...
}
```

The example above illustrates an interface design that returns an interface pointer to the caller correctly. See Part 4 of this paper for more details on the **iid_is** MIDL attribute.

In this example, the designer of the interface is allowing the caller to specify which interface he or she wants returned. This approach is preferred over forcing the interface to be **IUnknown**, as in the following version:

```
interface ISome2 : IUnknown
{
    ...
    HRESULT CreateSubObject([out]IUnk** ppUnk);
    ...
}
```

The reason to avoid allowing only **IUnknown** to be returned is that the caller most likely will immediately want to use **QueryInterface** to find some other, more useful interface. This will most likely result in additional cross-process (or cross-machine!) calls. Thus, by letting the caller specify the interface in the call, you can optimize performance.

In another case where an interface pointer is being returned, it may be tempting to specifically force a particular interface. The following example illustrates this:

```
interface ISome3 : IUnknown
{
    ...
    HRESULT CreateSomeOtherObject([out]ISomeOther** ppSomeOther);
    ...
}
```

While this is certainly better than the **ISome2** case above—because in most cases only one round trip is necessary to get the caller hooked up to the "useful" interface—things start to break down if **ISomeOther** is improved and an **ISomeOther2** interface is introduced. Clients upgraded to use the **ISomeOther2** interface will have no choice but to make two round trips to the server, one to get the **ISomeOther** interface by calling **CreateSomeOtherObject** and another to **QueryInterface** for **ISomeOther2**.

Callbacks

COM interfaces should never use traditional Windows-based callbacks (method parameters that are pointers to functions). Instead, an "outgoing" interface should be used. Thought must be made regarding circular references. Using the connection-point interfaces is the recommended mechanism because they provide the most flexibility and extensibility. However, the connection-point interfaces are not simple by any stretch of the imagination, and some programmers may be turned off by their complexity, especially given that a properly designed **Advise**/**UnAdvise** mechanism (such as the one found in Uniform Data Transfer) can work just fine.

Enumerators

Often objects need to provide the capability to enumerate over sub-objects or other lists of entities. In these cases, use a COM enumerator (and **IEnumXXX** interface).

Differences in the In-Process Programming Model

The design of COM strives to provide complete *call location transparency* (also called *local/remote transparency* and *location transparency*). The basic philosophy is as follows: *Unless the caller explicitly wants to know, all calls appear to be in-process*.

There is one very esoteric situation where there is a difference in the programming model, depending on whether the object is running in-process or out-of-process (actually out-of-apartment):

- In-process calls are call-by-reference—when the callee writes through an out-parameter pointer, the value pointed to in the caller address space is updated immediately (even before the call returns).

- Cross-process calls are closer to call-by-value-result. When the callee writes through the out-pointer, the value in the caller address space is not actually updated immediately. It is updated (copied into the callee's address space) by the interface proxy when the call returns across the process boundary.

This difference would be important if the callee function first wrote through the out parameter, and then called back into the caller before returning. At this point the value back in the original caller's space would vary. In the in-process case, it would be the updated value. In the cross-process case, it would be the "old value," pending update by the return of the original call.

COM addresses this problem by specifying that functions only write through out parameter pointers immediately before returning. The specific rule is as follows: The out parameters may not be updated until the return, and, until then, should not be relied upon by the caller. If an interface design requires a way to set a value that the caller is guaranteed to see before the call returns, then an outgoing interface should be used to set it.

Part 4: Using MIDL as an Interface Design Tool

COM supports three kinds of marshalling: standard, custom, and handler. Standard marshalling is the most common because it is the easiest to implement, and in almost all cases is all you need. In standard marshalling, the designer of an interface supplies a proxy/stub dynamic-link library (DLL) that implements an interface proxy and an interface stub for the interface. Both the interface proxy and interface stub are in-process COM objects that use a set of services (default interface implementations) provided by COM's standard marshalling layer. The COM standard marshalling layer is based on Microsoft RPC, which is compatible with DCE RPC. Readers familiar with Microsoft RPC (or DCE RPC) know that, when programming in RPC, you describe your RPC interfaces in a language called IDL and generate stubs using the IDL compiler. Microsoft has extended its IDL compiler (MIDL) to support the generation of COM-compatible interface proxy and stub objects. In this section we discuss using MIDL for the design of COM interfaces.

MIDL Syntax

The MIDL compiler can be used to generate standard DCE-compatible RPC stubs, RPC stubs that support the Microsoft extensions to DCE RPC (Microsoft RPC), or COM interface proxies and stubs. The following discussion is an overview of the IDL syntax used when dealing with COM interfaces. The Platform SDK contains full documentation for the MIDL compiler and the IDL syntax.

Below is an .IDL file for a COM interface that uses the most common IDL language elements:

```
// BARF.IDL

import "unknwn.idl" //0

[
object, //1
uuid(15d39410-f1e7-11ce-9055-080036f12502) //2
]
interface IBarf : IUnknown //3
{
    HRESULT MethodOne([in] ULONG ul); //4
    HRESULT MethodTwo([out] short* s); //5
    HRESULT MethodThree([in] REFIID riid, [out, iid_is(riid)] IUnknown** ppUnk);
                                                                        //6
}
```

The import directive (//0) causes the preprocessor to bring in all of the definitions found in the UNKNWN.IDL file. This file, which is included in the INCLUDE directory of the Platform SDK, contains the IDL descriptions for **IUnknown**, **IClassFactory**, and all of the COM base types. If your interface was derived from another, higher-level interface such as **IPersist**, you would want to import

OBJIDL.IDL instead. The **object** attribute (//2) indicates that this is a COM interface rather than an RPC interface, and allows interface definitions to be inherited (//3). The **uuid** attribute is where we specify our interface's IID (//1).

Within the curly brackets of the interface definition are listed the methods of that interface (//4). The order they appear here is the order they appear in the vtable. Each of the three methods in this example are illustrative. **MethodOne** utilizes the **in** attribute, which is the default if no attributes are specified. **MethodTwo** (//5) illustrates a method with an out parameter. **MethodThree** (//6) illustrates the common COM construct of an out parameter that is a pointer to a specific interface.

The IDL attributes that are commonly used for COM interfaces are **in**, **out**, **iid_is**, **size_is**, **length_is**, **string**, **unique**, **local**, and **call_as**, although other IDL attributes can be used.

Near the end of this section is a description of all the steps involved in running the MIDL compiler to generate a proxy/stub DLL from the BARF.IDL given above.

MIDL Tips

Add [v1_enum] on all enums

v1_enum means that 32 bits are sent (rather then 16) on the wire for the **enum**. This keeps structures that contain **enums** naturally aligned.

Don't overuse length_is

size_is is usually enough.

Avoid full [ptr] pointers

Full pointers **[ptr]** let you have aliasing between pointers to the same types. This approach is expensive, because it requires that a dictionary of all pointers marshalled (and unmarshalled) be maintained. It is likely that such data types can and will be avoided without significant cost when the designer is aware of the overhead.

Building a Proxy/Stub DLL

The process of building a proxy/stub DLL is extremely simple once you've written your .IDL file. The steps involved are:

1. Create a .DEF file that includes export information for the standard in-process COM object DLL exports.
2. Create a resource script file (.RC) that contains a VERSIONINFO resource.
3. Create a MAKEFILE that runs MIDL, compiles the generated C code, runs the RC compiler, and then links everything together.
4. Execute the MAKEFILE.

A sample BARF.DEF, BARF.RC, and MAKEFILE are given below, which you can cut and paste for your own interfaces.

Sample .RC File

The primary reason you want to include resources in your proxy/stub DLL is for the **OLESelfRegister** VERSIONINFO string. By including this string in your resources you indicate to the world that you are a self-registering COM server.

```
// BARF.RC
#include <windows.h>
#include <winver.h>

VS_VERSION_INFO    VERSIONINFO
  FILEVERSION      1,0,0,1
  PRODUCTVERSION   1,0,0,1
  FILEFLAGSMASK    VS_FFI_FILEFLAGSMASK
#ifdef _DEBUG
  FILEFLAGS        VS_FF_DEBUG|VS_FF_PRIVATEBUILD|VS_FF_PRERELEASE
```

```
#else
  FILEFLAGS        0 // final version
#endif
  FILEOS           VOS__WINDOWS32
  FILETYPE         VFT_DLL
  FILESUBTYPE      0   // not used
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "040904E4" // Lang=US English, CharSet=Windows Multilingual
    BEGIN
    VALUE "CompanyName",     "\0"
    VALUE "FileDescription", "Barf Interfaces Proxy/Stub DLL\0"
    VALUE "FileVersion",     "1.0.000\0"
    VALUE "InternalName",    "BARF\0"
    VALUE "LegalCopyright",  "\0"
    VALUE "LegalTrademarks", "\0"
    VALUE "OriginalFilename","BARF.DLL\0"
    VALUE "ProductName",     "BARF\0"
    VALUE "ProductVersion",  "1.0.000\0"

    VALUE "OLESelfRegister", "\0" // New keyword

    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x409, 1252
  END
END
```

Sample .DEF file

```
; BARF.DEF
LIBRARY         BARF

EXPORTS        DllGetClassObject       PRIVATE
                DllCanUnloadNow        PRIVATE
                DllRegisterServer      PRIVATE
                DllUnRegisterServer    PRIVATE
```

Running MIDL

As noted earlier, given an .IDL file, the MIDL compiler generates source code files for COM interface proxies and stubs. MIDL is run from the command line and, when dealing with COM interfaces, the syntax used is:

```
MIDL /ms_ext /app_config /c_ext <name>.idl
```

In most cases you'll use only these command-line switches. The Platform SDK contains complete documentation on these and other MIDL command-line switches.

If the sample .IDL file above were named BARF.IDL, and we ran MIDL on it using the following command:

```
MIDL /ms_ext /app_config /c_ext barf.idl
```

MIDL would generate four files: DLLDATA.C, BARF_P.C, BARF_I.C, and BARF.H, which you compile and link together to generate the proxy/stub DLL. We discuss each of these files below.

DLLDATA.C

```
/*********************************************************
   DllData file -- generated by MIDL compiler

        DO NOT ALTER THIS FILE

   This file is regenerated by MIDL on every IDL file compile.

   To completely reconstruct this file, delete it and rerun MIDL
   on all the IDL files in this DLL, specifying this file for the
   /dlldata command line option

*********************************************************/
#include <rpcproxy.h>
#ifdef __cplusplus
extern "C"   {
#endif
EXTERN_PROXY_FILE( barf )

PROXYFILE_LIST_START
/* Start of list */
  REFERENCE_PROXY_FILE( barf ),
/* End of list */
PROXYFILE_LIST_END

DLLDATA_ROUTINES( aProxyFileList, GET_DLL_CLSID )

#ifdef __cplusplus
}  /*extern "C" */
#endif
/* end of generated dlldata file */
```

When this file is compiled by a C compiler using the **-DDREGISTER_PROXY_DLL** command-line switch and linked to RPCRT4.LIB, the DLL will automatically have the **DllMain**, **DllGetClassObject**, **DllCanUnloadNow**, **DllRegisterServer**, and **DllUnRegisterServer** exports that every COM in-process server requires.

BARF_P.C

The <NAME>_P.C file generated by MIDL contains the source code for the interface proxy and stubs. You should never have to modify any of this code.

BARF_I.C

This file contains interface information, such as the structure containing the IID. This file can be linked into any program that requires your IID.

```
/* this ALWAYS GENERATED file contains the actual definitions of */
/* the IIDs and CLSIDs */

/* link this file in with the server and any clients */

/* File created by MIDL compiler version 2.00.0102 */
/* at Mon Sep 18 14:52:36 1995
 */
//@@MIDL_FILE_HEADING(  )
#ifdef __cplusplus
extern "C"{
#endif

#ifndef IID_DEFINED
#define IID_DEFINED

typedef struct _IID
{
    unsigned long x;
    unsigned short s1;
    unsigned short s2;
    unsigned char  c[8];
} IID;

#endif // IID_DEFINED

#ifndef CLSID_DEFINED
#define CLSID_DEFINED
typedef IID CLSID;
#endif // CLSID_DEFINED

const IID IID_IBarf = {0x15d39410,0xf1e7,0x11ce,{0x90,0x55,0x08,0x00,0x36,0xf1,0x25,0x02}};

#ifdef __cplusplus
}
#endif
```

BARF.H

The <NAME>.H file generated by MIDL is the header file for your interface. It contains definitions that can be used by either C or C++ code. For C programmers, MIDL generates handy macros that let you hide dealing with vtable pointers (see the **COBJMACROS** section). Note that the sample presented below has been edited to be more readable.

```
/* this ALWAYS GENERATED file contains the definitions for the interfaces */

/* File created by MIDL compiler version 2.00.0102 */
/* at Mon Sep 18 15:11:40 1995
 */
```

```
//@@MIDL_FILE_HEADING(  )
#include "rpc.h"
#include "rpcndr.h"
#ifndef COM_NO_WINDOWS_H
#include "windows.h"
#include "ole2.h"
#endif /*COM_NO_WINDOWS_H*/

#ifndef __barf_h__
#define __barf_h__

#ifdef __cplusplus
extern "C"{
#endif

... Stuff removed for readability ...


#if defined(__cplusplus) && !defined(CINTERFACE)
    interface IBarf : public IUnknown
    {
    public:
        virtual HRESULT __stdcall MethodOne(
            /* [in] */ ULONG ul) = 0;

        virtual HRESULT __stdcall MethodTwo(
            /* [out] */ short __RPC_FAR *s) = 0;

        virtual HRESULT __stdcall MethodThree(
            /* [in] */ REFIID riid,
            /* [iid_is][out] */ IUnknown __RPC_FAR *__RPC_FAR *ppUnk) = 0;
   };

#else     /* C style interface */
    typedef struct IBarfVtbl
    {
        HRESULT ( __stdcall __RPC_FAR *QueryInterface )(
            IBarf __RPC_FAR * This,
            /* [in] */ REFIID riid,
            /* [out] */ void __RPC_FAR *__RPC_FAR *ppvObject);

        ULONG ( __stdcall __RPC_FAR *AddRef )(
            IBarf __RPC_FAR * This);

        ULONG ( __stdcall __RPC_FAR *Release )(
            IBarf __RPC_FAR * This);

        HRESULT ( __stdcall __RPC_FAR *MethodOne )(
            IBarf __RPC_FAR * This,
            /* [in] */ ULONG ul);

        HRESULT ( __stdcall __RPC_FAR *MethodTwo )(
            IBarf __RPC_FAR * This,
            /* [out] */ short __RPC_FAR *s);

        HRESULT ( __stdcall __RPC_FAR *MethodThree )(
            IBarf __RPC_FAR * This,
```

```
            /* [in] */ REFIID riid,
            /* [iid_is][out] */ IUnknown __RPC_FAR *__RPC_FAR *ppUnk);
    } IBarfVtbl;

    interface IBarf
    {
        CONST_VTBL struct IBarfVtbl __RPC_FAR *lpVtbl;
    };

#ifdef COBJMACROS
#define IBarf_QueryInterface(This,riid,ppvObject)    \
    (This)->lpVtbl -> QueryInterface(This,riid,ppvObject)
#define IBarf_AddRef(This)    \
    (This)->lpVtbl -> AddRef(This)
#define IBarf_Release(This)    \
    (This)->lpVtbl -> Release(This)
#define IBarf_MethodOne(This,ul)    \
    (This)->lpVtbl -> MethodOne(This,ul)
#define IBarf_MethodTwo(This,s)    \
    (This)->lpVtbl -> MethodTwo(This,s)
#define IBarf_MethodThree(This,riid,ppUnk)    \
    (This)->lpVtbl -> MethodThree(This,riid,ppUnk)
#endif /* COBJMACROS */

#endif     /* C style interface */

... Stuff removed for readability ...


#ifdef __cplusplus
}
#endif
#endif
```

Sample makefile

```
#
# MAKEFILE
# Builds Proxy/Stub DLL for BARF.IDL
#
# Copyright (c) 1993-1995 Microsoft Corporation, All Rights Reserved

TARGET  = barf
OUTFILE = $(TARGET).dll

OBJS1   = barf_i.obj barf_p.obj
OBJS2   = dlldata.obj
OBJS    = $(OBJS1) $(OBJS2)

LIBS    = rpcrt4.lib

barf.dll: barf_i.obj barf_p.obj dlldata.obj barf.def barf.res
    link /DLL -def:$(TARGET).DEF -out:$(OUTFILE) $(OBJS) $(LIBS) $(TARGET).RES
```

```
barf_i.obj : barf_i.c barf.idl
    cl -MD -c barf_i.c

barf_p.obj : barf_p.c barf.h barf.idl
    cl -MD -c barf_p.c

dlldata.obj : dlldata.c barf.idl
    cl -MD -c -DREGISTER_PROXY_DLL dlldata.c

barf.res : barf.rc
    rc /r $(TARGET).rc

#Run MIDL.EXE to produce the source code
barf.h barf_p.c barf_i.c dlldata.c: barf.idl
    midl /ms_ext /app_config /c_ext $(TARGET).idl
```

To build the proxy/stub DLL for **IBarf**, simply type **NMAKE** at the command prompt.

Because BARF.DLL is a self-registering COM server, it can be registered via the REGSRV32 tool, negating the need for a separate .REG file.