

4. The Component Object Model

With this chapter we begin the presentation in this specification of the actual functions and interfaces that form OLE 2 and its underlying foundations. Detailed function prototypes and descriptions are presented.

The body of functionality discussed in this chapter forms the core of what is known as the "Component Object Model." This object model and the infrastructure that supports it forms the heart of the central underpinnings on which OLE 2 is built; however the model has nothing directly to do with compound documents *per se;* rather, it has a much broader applicability to component software in general.

The Component Object Model defines:

- the concept of an *interface* as the *lingua-franca* by which clients of services communicate with entities which provide those services. These entities are referred to as "components" or "objects." Interfaces promote clarity of understanding as to the services from a given component that a client can actually rely on using long-term as the component is revised, updated, and otherwise replaced.
- an architecture (QueryInterface) by which objects can support *multiple interfaces*, which provides a robust means for an component to support optional or new interfaces without breaking existing clients, and conversely provides a ubiquitous means by which a updated client, aware of new functionality it desires to use, can safely inquire of a given component whether it *too* understands this new functionality. This inquiry is safe even if the component is an old, unaware one.
- a *reference counting* model of object management that enables objects to be reclaimed when they are no longer needed, and which permits an object to be simultaneously used by many non-communicating clients, yet still know when it is no longer needed.
- a mechanism by which pointers to interfaces can be passed through *remote procedure calls*, enabling clients in other processes to access and manipulate objects. This mechanism is transparent to both the client and the server side of the conversation: servers need take no special action to be remoted, and clients need take no special action in order to deal with remoted services.
- an architecture (CoGetClassObject() and related infrastructure) by which a desired object implementation can be identified and *dynamically loaded* into the running system, either as a .DLL or an .EXE server in a separate process. Clients need not be aware of whether an object is implemented locally or remotely.
- a level of indirection to object implementations though the System Registry (see CoGetClass-Object()). Since static binding to the implementation (using DLL names, for example) is not used, this additional level of indirection frees clients from knowing the packaging into files (DLLs, EXEs) of the functionality they use from various services, allowing that packaging to be changed in the future without breaking existing clients.
- a standard mechanism (use of the memory allocator returned by CoGetMalloc()) by which ownership of data containing memory allocations can be passed between parties. To give but one motivating example, this allow strings to be returned from server to client without the client having to allocate a buffer in advance. Of course, this applies to any data value.

This chapter is organized as follows. We begin by examining the technical calling conventions for OLE interfaces used with both C and C++. Next, we discuss the stylistic convention used for returning function results and error status. Following this, IUnknown interface and enumerators are presented, then a discussion of memory management. A discussion of class binding and object instantiation is next, and we conclude with a look at part of the new remoting infrastructure put in place for OLE 2.

4.1. Calling conventions; Calling interfaces from C++, C, or other languages

As was mentioned in the previous chapter, OLE 2 interfaces are documented in this specification using C++ syntax as a notation, and programmers using C++ compilers will be able to call these functions directly and conveniently. However, it is by no means required that programmers use C++, or any other particular language for that matter: the Component Object Model is based on a *binary* interoperability standard, rather than a language-based interoperability standard, as are some other systems. While the model is more easily used in some languages than others, in principle any language supporting "struct" or "record" types containing double-indirected access to a table of function pointers is suitable.

That being said, the OLE 2 product contains interface declarations for both C++ and C, but not for other languages; see the header file COMPOBJ.H and references to the macro CINTERFACE in particular for details on generating either the C++ or the C versions as needed.

Given the C++ definition of an interface, which in general is of the form:

```
interface ISomeInterface {
    virtual     RET_T    MemberFunction(ARG1_T arg1, ARG2_T arg2 /*, etc */);
    // usually other member functions, too.
    };
```

then the corresponding C declaration of that interface looks like

```
typedef struct ISomeInterface {
    ISomeInterfaceVtbl*  vptr;
    } ISomeInterface;

typedef struct ISomeInterfaceVtbl    ISomeInterfaceVtbl;

struct ISomeInterfaceVtbl {
    RET_T (*MemberFunction)(ISomeInterface * This, ARG1_T arg1, ARG2_T arg2 /*, etc */);
    } ;
```

This example also illustrates the algorithm for determining the signature of C form of an interface function given the corresponding C++ form of the interface function:

- Use the same argument list as that of the member function, but add an initial parameter which is the pointer to the interface. This initial parameter is a pointer to a C type of the same name as the interface.
- A struct type which is a table of function pointers corresponding to the vtbl layout of the interface exists as the name of the interface followed by "Vtbl". Members in this struct have the same names as the member functions of the interface.

All OLE 2 member functions use the "far cdecl" calling convention: "far" indicates a 32-bit return address; "cdecl" indicates that the arguments are pushed right-to-left, so the first argument is closest to the return address, and that the caller is responsible for popping the arguments. (This is nothing new; "far cdecl" has for a long time had these semantics in MS-DOS compilers.) When calling member functions, we need also to define the positioning of the hidden argument which is a pointer to the object instance. The Microsoft Object Mapping[30] specifies that this pointer is pushed very last, immediately before the return address. The location of this pointer is the reason that the pintf pointer appears at the *beginning* of the argument list of the equivalent mapping functions: it means that the layout in the stack of the parameters to the mapping function is exactly that expected by the member function, and so no re-ordering is required.[31]

OLE 2 API functions (in contrast to *interface member* functions) use the "far pascal" calling convention.

---

[30] The "Microsoft Object Mapping" is an open specification describing the detailed layout of C++ objects. It is supported the MS C/C++ compiler, as well as C++ compilers from other vendors including Borland.

[31] Use of the "pascal" convention, rather than "cdecl", was considered, since it has the space efficiency of the callee, not the caller, popping the arguments. However, "pascal" calling convention also implies a left-to-right pushing order for the arguments. Since on 32-bit MS-DOS compilers no standard convention exists which supports left-to-right evaluation, and right-to-left vs. left-to-right determines whether the object pointer goes first or last in the mapping function, in order to maintain source compatibility we are forced even in the 16 bit world to use right-to-left. The only right-to-left 16-bit convention is cdecl.

It is further the case that all pointers passed through OLE interfaces or APIs are far pointers (32 bit); no near pointers (16 bit) are used.[32] As a consequence, it is often the case that in this specification, the keyword "far" is omitted for brevity; it should be inferred by the reader. The header files that ship with the OLE product are, of course, precise about this.

Finally, this specification makes some use of the C++ reference mechanism in parameter passing, for example QueryInterface(const IID FAR& iidInterface, void** ppvObj). As you might expect, this example will appear in C version of the interface as a parameter of type const IID FAR* const. Input parameters passed by reference will themselves be const, as shown here. In-out or output parameters will not.

OLE 2, 32 bit, will use slightly different calling conventions than is described here, but will remain source-compatible with 16 bit OLE 2.

4.2. Returning values & Reporting error status

OLE 2 interface member functions and API functions use a certain stylistic convention in order to pass back to the caller both a useful return value and / or an indication of status or error information. The general form of this convention is:

```
typedef void *   HRESULT;
typedef LONG     SCODE;
#define          NOERROR     0

SCODE        GetScode(HRESULT hresult);
HRESULT      ResultFromScode(SCODE scode);
HRESULT      PropagateResult(hrPrev, scNew);

HRESULT      SomeFunction(ARG1_T arg1, ... , ARGN_T argn, RET_T * pret);
```

That is, the return value is an opaque "result handle" of type HRESULT, and would otherwise be passed as the return value in the absence of the need for status information is passed through a pointer as the last argument. HRESULT values are defined to be zero for success, and non-zero if error or informational status is being returned. If the result value is non-zero, the application calls GetScode() to map the HRESULT into a known code. For convenience, GetScode() is defined to return S_OK (see below) for a zero hresult.

On the implementor's side, an interface member function that wants to indicate a result other than simple success must call ResultFromScode() to generate the corresponding HRESULT, and return that to its caller. For convenience, ResultFromScode() is defined to return zero for an scode of S_OK.; NOERROR is a constant of type HRESULT that corresponds to S_OK.

HRESULTs do not need to be explicitly freed; there is, therefore, no function "FreeHresult" or the like.

This may appear to be a long-winded way of returning a simple SCODE value; the intent is to allow us to define interfaces that, over time, can be used with components that send extended error information out of band, for use in problem diagnosis and recovery.

Sometimes we run across functions that wish either to return a Boolean result, or, if the result is not currently available, to be able to return an error status. For these we frequently adopt the special convention of using two particular status codes, S_OK and S_FALSE to indicate the return value. This obviates the need for a separate BOOL* parameter for these functions.

A status code is a 32-bit value of type SCODE which is divided up into four fields. The following is the format of an SCODE, where the numbers indicate bit positions:

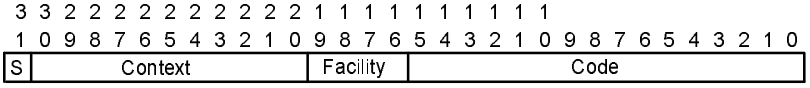


Figure 69. Structure of an SCODE

The fields in an SCODE have the following semantics.

---

[32] Actually, when significant, these are "huge pointers" that can point to data blocks larger than 64k.

S: Severity field. The following values are defined for this field:

0 *Success*. This value means that the function was successful.

1 *Error*. This value means that the function incurred an error.

Context: Reserved for future use; must be set to zero by present programs generating SCODEs; however, present code should *never* examine this field, as it may be set as non-zero in the implementation of PropagateResult() in future system releases.

Facility: Indicates which group of status codes this belongs to. New facilities are allocated by Microsoft, since they need to be unique. However, in many cases FACILITY_ITF will be adequate; see below.

Code: This field describes what actually took place.

In OLE 2, the following SCODE facilities are used.

Facility	Description
FACILITY_NULL	This facility is used for broadly applicable common status codes. S_OK belongs to this facility, for example. The value for this facility code is zero.
FACILITY_RPC	This facility is used for errors that result from an underlying remote procedure call implementation. In general, in this specification we do not explicitly document the RPC errors that can be returned from functions, though they nevertheless can be returned in situations where the interface being used is in fact remoted.
FACILITY_STORAGE	This facility is used for persistent-storage-related errors. Status codes whose code (lower 16 bits) value is in the range of DOS error codes have the same meaning as the corresponding DOS error.
FACILITY_DISPATCH	This facility is used for late binding (IDispatch) errors.
FACILITY_ITF	This facility is used for most status codes that are returned from an interface member function. Use of this facility indicates that the meaning of the error code is defined solely by the definition of the particular interface in question; an SCODE with exactly the same 32-bit value returned from another interface might have a different meaning.

By convention, SCODEs generally have names in the following form:

<Facility>_<Sev>_<Reason>

where **<Facility>** is either the facility name or some other distinguishing identifier, **<Sev>** is a single letter, one of the set { S, E } indicating the severity, and **<Reason>** is a short identifier that describes the meaning of the code. Status codes from FACILITY_NULL omit the **<Facility>**_ prefix. For example, the status code STG_E_FILENOTFOUND indicates, as its name might suggest, that a storage-related error has occurred; specifically, a file that was requested does not exist.

The use of FACILITY_ITF deserves some special discussion with respect to interfaces defined in OLE2 and interfaces that will be defined in the future. Where as SCODEs with other facilities (FACILITY_NULL, FACILITY_RPC, etc.) have universal meaning, SCODEs in FACILITY_ITF have their meaning completely determined by the interface member function (or API function) from which they are returned; the same 32-bit value in FACILITY_ITF returned from two different interface functions may have completely different meanings. The reasoning behind this distinction is as follows. For reasons of efficiency, it is unreasonable to have the primary error code data type (SCODE) be larger than 32 bits in size. 32 bits, unfortunately, is not large enough to enable us to develop an allocation policy for error codes that will universally avoid conflict between codes allocated by different non-communicating programmers at different times in different places (contrast, for instance, with what we do with IIDs and CLSIDs). Therefore, we have structured the use of the 32 bit SCODE in such a way so as to allow Microsoft to define *some* universally

defined error codes while at the same time allowing other programmers to define new error codes without fear of conflict by limiting the places in which those field-defined error codes can be used. Thus:

1. Status codes in facilities other than FACILITY_ITF can only be defined by Microsoft.
2. Status codes in facility FACILITY_ITF are defined solely by the *definer of the interface* or API by which said status code is returned. That is, in order to avoid conflicting error codes, a human being needs to coordinate the assignment of codes in this facility, and we state that he who defines the interface gets to do the coordination.

OLE2 itself defines many interfaces and APIs, and so OLE2 defines many status codes in FACILITY_ITF; see the header file SCODE.H for details. By design, none of the OLE2-defined status codes in fact have the same value, even if returned by different interfaces, though it would have been legal for OLE2 to do otherwise. With regard to which errors can be returned by which functions, it is the case that, in the extreme,

- It is legal that any OLE2-defined status code may in fact be returned by any OLE2-defined interface member function or API function.
- Further, any error in FACILITY_RPC or FACILITY_DISPATCH, even those not presently defined, may be returned.
- Further, being the designer of the OLE interfaces (see point 2. above), we state that we reserve the right to in the future define new *failure* codes (but not success codes) for these interfaces, in FACILITY_ITF or new facilities; more below.

Normally, of course, only a small subset of the OLE2-defined status codes will be usefully returned by a given interface function or API, but the immediately preceding statements are in fact the actual interoperability rules for the OLE2 defined interfaces. In this specification, we have endeavored to point out which error codes are particularly useful for each function, and this is developed further in the appendix to the Technical Reference book, but code must be written to correctly handle the general rule. The present document is, however, precise as to which *success* codes may legally be returned.

Conversely, it is *only* legal to return a status code from the implementation of an interface member function which has been sanctioned by the designer of that interface as being legally returnable; otherwise, there is the possibility of conflict between these returned code values and the codes in-fact sanctioned by the interface designer. Pay particular attention to this when propagating errors from internally called functions. Nevertheless, callers of interfaces would be wise to guard themselves from imprecise interface implementations by treating any otherwise unknown returned error code (in contrast with success code) as synonymous with E_UNEXPECTED: it our experience that programmers are notoriously lax in dealing with error handling. Further, given the third bullet point above, this coding practice is *required* by clients of the OLE2-defined interfaces and APIs. Pragmatically speaking, however, this is little burden to programmers: normal practice is to handle a few special error codes specially, but treat the rest generically.

All the OLE-defined FACILITY_ITF codes in fact have a code value which lies in the region 0x0000 – 0x01FF. Thus, while it is indeed legal for the definer of a new function or interface to make use of any codes in FACILITY_ITF that he chooses in any way he sees fit, it is highly recommended that only code values in the range 0x0200 – 0xFFFF be used, as this will reduce the possibility of accidental confusion with any OLE-defined errors. It is also highly recommended that designers of new functions and interfaces consider defining as legal that most if not all of their functions can return the appropriate status codes defined by OLE2 in facilities other than FACILITY_ITF. In particular, interfaces that hold even the remote possibility of ever being remoted using RPC at some time in the future should define the FACILITY_RPC codes as legal. E_UNEXPECTED is a specific error code that most if not all interface definers will wish to make universally legal.

Several functions exist that manipulate status codes:

4.2.0.1. SCODE_CODE

#define **SCODE_CODE**(sc)
Extract the error code part of the status code.

4.2.0.2. SCODE_FACILITY

```
#define SCODE_FACILITY(sc)
```

Extract the facility from the scode.

4.2.0.3. SCODE_SEVERITY

```
#define SCODE_SEVERITY(sc)
#define SEVERITY_SUCCESS        0
#define SEVERITY_ERROR          1
```

Extract the severity field from the scode.

4.2.0.4. SUCCEEDED

```
#define SUCCEEDED(Status)
```

Returns true if the severity of the status code is success, false otherwise.

4.2.0.5. FAILED

```
#define FAILED(Status)
```

Returns true if the severity of the status code is error, false otherwise.

4.2.0.6. MAKE_SCODE

```
#define MAKE_SCODE(sev,fac,code)
```

Make a new scode given a severity, a facility, and a code.

4.2.0.7. GetScode

SCODE GetScode(hresult)

Extract the SCODE contained in an HRESULT.

Argument	Type	Description
hresult	HRESULT	the HRESULT from which the SCODE is to be extracted
return value	SCODE	the extracted SCODE

4.2.0.8. ResultFromScode

HRESULT ResultFromScode(scode)

Generate a new HRESULT that contains the given scode.

This method of generating an HRESULT should be used if an is being error returned as a result of some internal state error. By contrast, it should not be used if the error is being returned as a result of some internally called routine itself returning an error HRESULT. In the latter case, the calling routine should instead use PropagateResult() to return its error.

In the present release of OLE, HRESULTs do little more than wrap the SCODE they are given. However, in a future system release, the implementation of HRESULT will be transparently enhanced so as to convey more information as to the nature of the error and where it occurred.[33] Using ResultFromScode() where PropagateResult() is more correctly used will not in any way be fatal, but will prevent this future enhancement to HRESULTs from achieving its full benefit. Please use the two routines correctly.

Argument	Type	Description
scode	SCODE	the SCODE to be encapsulated in an HRESULT.
return value	HRESULT	the new HRESULT that encapsulates it.

[33] Yes, we know how to do this. It's a little tricky, but can indeed really be done transparently.

4.2.0.9. PropagateResult

HRESULT PropagateResult(hrPrev, scNew)

This function is used to generate an HRESULT to return to a function's caller in the event that the error being returned was in fact caused by some internally called function itself returning an error HRESULT. In future implementations, this call will build up a system-maintained stack of error-result contexts that will be used heuristically in Product Support tools. Merely returning the internally generated HRESULT instead of calling PropagateResult() will never be fatal, but future support tools will not be as capable as they otherwise would be. Further, one must always be careful to only return error codes through an interface that are permitted as was specified by the designer of that interface.

Argument	Type	Description
hrPrev	HRESULT	the HRESULT returned from the internally called routine.
scNew	SCODE	the new SCODE to return to *our* caller, wrapped in an HRESULT.
return value	HRESULT	the (new) HRESULT that we should return to our caller.

4.3. IUnknown interface

IUnknown supports the capability of getting to other interfaces on the same object through QueryInterface(). In addition, it supports the management of the existence of the interface instance though Release() and AddRef(). There are many other interfaces in the system[34] which derive from IUnknown; that is, they have the member functions of IUnknown as a prefix of their own; their vtbl begins with the vtbl entries of IUnknown.

```
interface IUnknown {
    virtual     HRESULT     QueryInterface(iidInterface, ppvObj) = 0;
    virtual     ULONG       AddRef() = 0;
    virtual     ULONG       Release() = 0;
    };
```

4.3.0.1. IUnknown::QueryInterface

HRESULT IUnknown::QueryInterface(iidInterface, ppvObject)

Return a pointer within this object instance that implements the named interface. Answer NULL if the receiver does not contain an implementation of the interface. In OLE 1, this function was named "QueryProtocol()". The semantic functionality has been preserved; only the name has been changed.

It is required that any query for the specific interface "IUnknown" (by calling QueryInterface(IID_IUnknown, ...)) always returns the *same actual pointer value*, no matter from which interface derived from IUnknown it is called. This enables the following algorithm to determine whether two pointers in fact point to the same object: call QueryInterface(IID_IUnknown, ...) on both and compare the results.

In contrast, queries for interfaces *other* than IUnknown are *not* required to return the same actual pointer value each time a QueryInterface() returning one of them is called.

It is required that the set of interfaces accessible on an object via QueryInterface() be static, not dynamic, in the following precise sense. It is necessary that we be able to rely on this set of rules in order to be able to provide remote access to interface pointers with a reasonable degree of efficiency.

Suppose we have a pointer to an interface

```
ISomeInterface * psome = (some function returning an ISomeInterface *);
```

where ISomeInterface derives from IUnknown. Suppose further that the following operation is attempted:

```
IOtherInterface * pother;
SCODE sc = psome->QueryInterface(IID_IOtherInterface, &pother);          // line 3
```

[34] In fact, almost all of them.

Then, the following must be true:

- If sc==S_OK, then if the QueryInterface() in line 3 is attempted a second time from the same psome pointer, then S_OK must be answered again. This is independent of whether pother was Released() in the interim. In short, if you can get to a pointer once, you can get to it again.
- If sc==E_NOINTERFACE, then if the QueryInterface() in line 3 is attempted a second time from the same psome pointer, then E_NOINTERFACE must be answered again. In short, if you didn't get it the first time, then you won't get it later.

Furthermore, QueryInterface() must be reflexive, symmetric, and transitive with respect to the set of interfaces that are accessible. That is, given the above definitions, then we have the following:

Symmetric: psome->QueryInterface(IID_ISomeInterface, ...) must succeed

Reflexive: If in line 3, pother was successfully obtained, then

```
pother->QueryInterface(IID_ISomeInterface, ...)
```

must succeed.

Transitive: If in line 3, pother was successfully obtained, and we do

```
IYetAnother * pyet;
SCODE sc2 = pother->QueryInterface(IID_IYetAnother, &pyet);          // line 6
```

and pyet is successfully obtained in line 6, then

```
pyet->QueryInterface(IID_ISomeInterface, ...)
```

must succeed.

Here, "must succeed" means "must succeed barring catastrophic failures." It is specifically *not* the case that two QueryInterface()s on the same pointer asking for the same interface must succeed and return exactly the same *pointer value*;[35] all that we require is that the QueryInterface() not say "sorry, that interface is not available."

Argument	Type	Description
iidInterface	REFIID	the interface desired.
ppvObj	void **	pointer to the object with the desired interface. In the case that the interface is not supported (i.e.: E_NOINTERFACE is returned) *ppvObj must be set to NULL.
return value	HRESULT	S_OK if the interface is supported, E_NOINTERFACE if it is not.

4.3.0.2. IUnknown::Release

ULONG IUnknown::Release()

Release a reference to this object. If AddRef() has been called on this interface n times and this is the n+1 th call to Release(), then the interface will free itself. If that freed interface was the only extant interface on an object when supports multiple interfaces (through QueryInterface(), of course), then the object will free itself.

Argument	Type	Description
return value	ULONG	The resulting value of the reference count. This value is returned solely for diagnostic / testing purposes; it absolutely cannot be used by shipping code, since in certain situations it is unstable. Release cannot indicate failure; if a client needs to know that resources have been freed etc., it must use a method in some interface on the object with higher level semantics before calling release.[36]

[35] With the single exception if QueryInterface(IID_IUnknown, ...) as was previously discussed.

[36] Due to present OLE2 implementation restrictions, it is required that Release() return zero if and only if the identity of the object has just been destroyed; that is, if the object has gone away. This restriction *will* be removed in future implementations; clients absolutely cannot rely on zero having a special significance other than for debugging purposes.

4.3.0.3. IUnknown::AddRef

ULONG IUnknown::AddRef()

May be used by clients when copying a pointer to the interface in situations where the cloned pointer's lifetime must extend beyond the Release() through the original pointer. In such situations the cloned pointer must also be released by invoking the Release() method.

Objects implementations are required to support a certain minimum size for the counter that is internally maintained by AddRef(). In short, this counter must be at least 31 bits large. The precise rule is that the counter must be large enough to support 2^{31}-1 outstanding pointer references to all the interfaces on a given object taken as a whole. Just make it a 32 bit unsigned integer, and you'll be fine.

Argument	Type	Description
return value	ULONG	The resulting value of the reference count. This value is returned solely for diagnostic / testing purposes; it absolutely cannot be used by shipping code, since in certain situations it is unstable.

4.3.0.4. Reference Counting

Objects accessed through interfaces use a reference counting mechanism to ensure that the lifetime of the object includes the lifetime of references to it. This mechanism was adopted so that independent components can obtain and release access to a single object, and not have to coordinate with each other over the lifetime management. In a sense, the object provides this management, so long as the client components conform to the rules. Within a single component that is completely under the control of a single development organization, clearly that organization can adopt whatever strategy it chooses. The following rules are about how to manage and communicate interface instances between components, and are a reasonable starting point for a policy within a component.

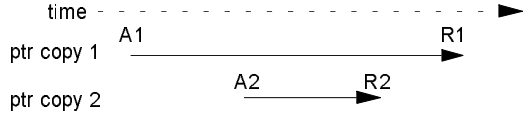
Note that the reference counting paradigm applies only to pointers to interfaces; pointers to data are not referenced counted.

It is important to be very clear on exactly when it is necessary to AddRef() and Release() an interface pointer. By its nature, pointer management is a cooperative effort between separate pieces of code, which must all therefore cooperate in order that the overall management of the pointer be correct. The following discussion should hopefully clarify the rules as to when AddRef() and Release() need to be done in order that this may happen. Some special reference counting rules apply to objects which are aggregated; see the discussion of aggregation in Chapter 3.

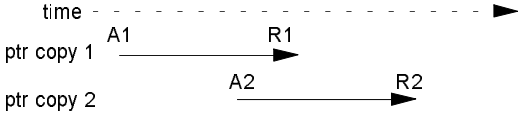
The conceptual model is the following. Interface pointers are thought of as living in pointer variables, which for the present discussion will include variables in memory locations and in internal processor registers, and will include both programmer- and compiler-generated variables. In short, it includes all internal computation state that holds an interface pointer. 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 is now two. An assignment to a pointer variable *destroys* the pointer copy presently in the variable, as does the destruction of the variable itself (i.e.: the scope in which the variable is found, such as the stack frame, is destroyed).

> **Rule 1**: Every new copy of an interface pointer must be AddRef()'d, and every destruction of an interface pointer must be Release()'d except where subsequent rules explicitly permit otherwise.

This is the default case. In short, unless special knowledge permits otherwise, the worst case must be assumed. The exceptions to Rule 1 all involve knowledge of the relationships of the lifetimes of two or more copies of an interface pointer. In general, they fall into two categories.



Category 1. Nested lifetimes



Category 2. Staggered overlapping lifetimes

In Category 1 situations, the AddRef() A2 and the Release() R2 can be omitted, while in Category 2, A2 and R1 can be elided.

Rule 2: Special knowledge on the part of a piece of code of the relationships of the beginnings and the endings of the lifetimes of two or more copies of an interface pointer can allow AddRef() / Release() pairs to be omitted.

The following rules call out specific common cases of Rule 2.

Rule 2a: *In-parameters to functions.* The copy of an interface pointer which is passed as an actual parameter to a function has a lifetime which is nested in that of the pointer used to initialize the value. The actual parameter therefore need not be separately reference counted.

Rule 2b: *Out-parameters from functions, including return values.* This is a Category 2 situation. In order to set the out parameter, the function itself by Rule 1 must have a stable copy of the interface pointer. On exit, the responsibility for releasing the pointer is transferred from the callee to the caller. The out parameter thus need not be separately reference counted.

Rule 2c: *Local variables.* A function implementation clearly has omniscient knowledge of the lifetimes of each of the pointer variables allocated on the stack frame. It can therefore use this knowledge to omit redundant AddRef() / Release() pairs.

Rule 2d: *Backpointers*. Some data structures are of the nature of containing two components, A and B, each with a pointer to the other. If the lifetime of one component (A) is known to contain the lifetime of the other (B), then the pointer from the second component back to the first (from B to A) need not be reference counted. (Often, avoiding the cycle that would otherwise be created is important in maintaining the appropriate freeing behaviour.)

The following rules call out common non-exceptions to Rule 1.

Rule 1a: *In-Out-parameters to functions.* The caller must AddRef() the actual parameter, since it will be Released() by the callee when the out-value is stored on top of it.

Rule 1b: *Fetching a global variable.* The local copy of the interface pointer fetched from an existing copy of the pointer in a global variable must be independently reference counted since called functions might destroy the copy in the global while the local copy is still alive.

Rule 1c: *New pointers synthesized out of "thin air".* A function which synthesizes an interface pointer using special internal knowledge rather than obtaining it from some other source must do an initial AddRef() on the newly synthesized pointer. Important examples of such routines include instance creation routines, implementations of IUnknown::QueryInterface(), etc.

Rule 1d: *Returning a copy of an internally stored pointer.* Once the pointer has been returned, the callee has no idea how its lifetime relates to that of the internally stored copy of the pointer. Thus, the callee must AddRef() the pointer copy before returning it.

Finally, when implementing or using reference counted objects, a technique sometimes termed "artificial reference counts" sometimes proves useful. Suppose you're writing the code in method `Foo()` in some interface Intf. If in the implementation of Foo() you invoke functions which have even the remotest chance of decrementing your reference count, then such function may cause you to release before it returns to `Foo()` The subsequent code in `Foo()` will crash.

A robust way to protect yourself from this is to insert an `AddRef()` at the beginning of `Foo()` which is paired with a `Release()` just before `Foo()` returns:

```
void Intf::Foo() {
    this[37]->AddRef();
    // Body of XXX, as before, except short-circuit returns need to be changed.
    this->Release();
    return;
    }
```

These "artificial" reference counts guarantee object stability while processing is done.

4.4. Enumerators

A frequent programming task is that of iterating through a sequence of items. The OLE 2 interfaces are no exception: there are places in several interfaces described in this specification where a client of some object needs to iterate through a sequence of items controlled by the object. We support such enumerations through the use of "enumerator objects." Enumerators cleanly separate the caller's desire to loop over a set of objects from the callee's knowledge of how to accomplish that function.

Enumerators are just a concept; there is no actual interface called IEnumerator or IEnum or the like. This is due to the fact that the function signatures in an enumerator interface must include the type of the things that the enumerator enumerates. As a consequence, separate interfaces exist for each kind of thing that can be enumerated: IEnumString, IEnumUnknown, etc. However, the difference in the type being enumerated is the *only* difference between each of these interfaces; they are all used in fundamentally the same way. Programming language people would speak of enumerators being "generic" over the element type. We shall take advantage of this here and document the semantics of enumerators as the interface "IEnum" using the C++ parameterized type syntax.

After discussing the generic IEnum interface, we follow by discussing concrete examples of enumeration interfaces so that readers may get a better idea of how they are used. In general, though, the actual enumeration interfaces used in OLE 2 are defined and documented in the interface members functions by which enumerations are instantiated.

```
template <class ELT_T> interface IEnum : IUnknown {
    virtual    HRESULT    Next(ULONG celt, ELT_T *rgelt, ULONG* pceltFetched) = 0;
    virtual    HRESULT    Skip(ULONG celt) = 0;
    virtual    HRESULT    Reset() = 0;
    virtual    HRESULT    Clone(IEnum<ELT_T>** ppenum) = 0;
    };
```

A typical use of an enumerator is the following.

```
interface StringContainer {
    virtual IEnumString* EnumStrings() = 0;
    };

void SomeFunc(StringContainer * pstringcont) {
    LPSTR lpsz;
    IEnumString* penum;
    penum = pstringcont->EnumStrings();
    while (penum->Next(1, &lpsz, NULL) == S_OK)
        {
        // do something with the string in lpsz
        // free the string in lpsz with the allocator returned by CoGetMalloc(MEMCTX_TASK, ...);
        }
    penum->Release();
    // penum is not valid here
    }
```

[37] "this" is the appropriate thing to AddRef() in an object implementation using the approach of multiply inheriting from the suite of interfaces supported by the object; more complex implementation strategies will need to modify this appropriately.

4.4.0.1. IEnum::Next

HRESULT IEnum::Next(celt, rgelt, pceltFetched)

Attempt to get the next celt items in the enumeration sequence, and return them through the array pointed to by rgelt. If fewer than the requested number of elements remain in the sequence, then just return the remaining ones; the actual number of elements returned is passed through *pceltFetched (unless it is NULL). If the requested celt elements are in fact returned, then return S_OK; otherwise return S_FALSE. An error condition other than simply "not that many elements left" will return an SCODE which is a failure code rather than one of these two success values.

To clarify:

- If S_OK is returned, then on exit the all celt elements requested are valid and returned in rgelt.
- If S_FALSE is returned, then on exit only the first *pceltFetched entries of rgelt are valid. The contents of the remaining entries in the rgelt array are indeterminate.
- If an error value is returned, then on exit no entries in the rgelt array are valid; they are *all* in an indeterminate state.

Argument	Type	Description
celt	ULONG	the number of elements that are to be returned.
rgelt[38]	ELT_T*	an array of size at least celt in which the next elements are to be returned.
pceltFetched	ULONG*	May be NULL if `celt==1`. If non-NULL, then this is set with the number of elements actually returned in rgelt.
return value	HRESULT	S_OK if the number of elements returned is celt; S_FALSE if fewer are returned.

4.4.0.2. IEnum::Skip

HRESULT IEnum::Skip(celt)

Attempt to skip over the next celt elements in the enumeration sequence. Return S_OK if this was accomplished, or S_FALSE if the end of the sequence was reached first.

Argument	Type	Description
celt	ULONG	the number of elements that are to be skipped.
return value	HRESULT	S_OK if the number of elements skipped is celt; S_FALSE otherwise.

4.4.0.3. IEnum::Reset

HRESULT IEnum::Reset()

Reset the enumeration sequence back to the beginning.

Note that there is no intrinsic guarantee that *exactly* the same set of objects will be enumerated the second time as was enumerated the first. Though clearly very desirable, whether this is the case or not is dependent on the collection being enumerated; some collections will simply find it too expensive to maintain this condition. Consider enumerating the files in a directory, for example.

Argument	Type	Description
return value	HRESULT	S_OK

[38] Think of "rgelt" as short for "range of elt", signifying an array.

4.4.0.4. IEnum::Clone

HRESULT IEnum::Clone(ppenum)

Return another enumerator which contains exactly the same enumeration state as this one. Using this function, a client can remember a particular point in the enumeration sequence, then return to it at a later time. Notice that the enumerator returned is of the same actual interface as the one which is being cloned.

Caveats similar to the ones found in IEnum::Reset() regarding enumerating the same sequence twice apply here as well.

Argument	Type	Description
ppenum	IEnum<ELT_T>**	the place in which to return the clone enumerator.
return value	HRESULT	S_OK.

4.4.1. Example: IEnumString interface

IEnumString is an enumerator which enumerates strings. Recall that LPSTR is the type that indicates a pointer to a zero terminated string of characters.

```
typedef IEnum<LPSTR>  IEnumString;
```

The immediately preceding typedef takes advantage of the C++ parameterized type IEnum<ELT_T> defined previously. The following is the equivalent expanded interface definition:

```
interface IEnumString: IUnknown {
    virtual     HRESULT     Next(ULONG celt, LPSTR *rgelt, ULONG* pceltFetched) = 0;
    virtual     HRESULT     Skip(ULONG celt) = 0;
    virtual     HRESULT     Reset() = 0;
    virtual     HRESULT     Clone(IEnumString** ppenum) = 0;
    };
```

We shall not describe each member function in detail here. The reader should instead refer to the above generic description of IEnum and substitute “LPSTR” for each occurrence of “ELT_T”.

4.4.2. Example: IEnumUnknown interface

IEnumUnknown is an enumerator which enumerates objects with interface IUnknown.

```
typedef IEnum<IUnknown*>    IEnumUnknown;
```

As before, the preceding takes advantage of the C++ parameterized type IEnum<ELT_T> defined above. The following is its equivalent expanded interface definition:

```
interface IEnumUnknown : IUnknown {
    virtual     HRESULT     Next(ULONG celt, IUnknown* rgelt, ULONG* pceltFetched) = 0;
    virtual     HRESULT     Skip(ULONG celt) = 0;
    virtual     HRESULT     Reset() = 0;
    virtual     HRESULT     Clone(IEnumUnknown ** ppenum) = 0;
    };
```

We shall not describe each member function in detail here. The reader should instead refer to the above generic description of IEnum and substitute “IUnknown*” for each occurrence of “ELT_T”.

Other enumerator interfaces are defined elsewhere in this specification on an as-needed basis in a completely analogous manner.

4.5. Memory management

Many, if not most, APIs and member functions in OLE 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; however, sometimes there arises the need to pass data structures for which this is not the case, and for which it is therefore necessary that the caller and the callee agree as to the allocation and de-allocation policy. This could in theory be decided and documented on an individual, function by function basis. However, it is much more reasonable to adopt a

universal convention for dealing with these parameters. Also, having a clear convention is important technically in order that the OLE 2 remote procedure call implementation can correctly manage memory.

Memory management of pointers to interfaces is always provided by member functions in the interface in question. For all the interfaces defined by OLE 2, these are the AddRef() and Release() functions found in the IUnknown interface, from which all other OLE 2 interfaces derive; see earlier in this chapter for a discussion of the reference counting supported by these functions. The rest of this section on memory management relates only to non-by-value parameters which are *not* pointers to interfaces, but are instead more mundane things like strings, pointers to structs, etc.

OLE makes use of two different kinds of memory: local application task memory, and shared (between process) memory, also called "global" memory. Each kind of memory has a different memory allocator implementation. All non-shared memory allocated by the OLE libraries or by object handlers is allocated using an application-supplied memory allocator passed as a parameter to CoInitialize(). If the application doesn't care to provide its own allocator, then it can pass NULL to CoInitialize(). This design gives the application control of how memory is actually allocated, while still allowing for the improved efficiency that results from often not having to allocate copies of data merely to pass as parameters to OLE functions. It also results in improved robustness in the face of application crashes, since the task memory is owned by an application and will be freed by Windows when the application terminates.

Shared memory is used much less frequently in OLE than is task memory. The primary use of shared memory is to optimize data copying that might otherwise need to occur in an RPC call: if the caller knows that the data it is allocating will be passed as parameter to a function that results in remote procedure call to another process, allocating it in shared memory to begin with will be more efficient. In OLE 2, "shared memory" specifically means memory that is allocated with the memory allocator returned by `CoGetMalloc(MEMCTX_SHARED, &pmalloc)`.

Each parameter to and the return value of a function can be classified into one of three groups: an **in** parameter, an **out** parameter, or an **in-out** parameter (the return value is treated as an out parameter). In each class of parameter, the responsibility for allocating and freeing non-by-value parameters is the following:

in parameter: These are both allocated and freed by the caller.

out parameter: These are to be allocated by the callee, but freed by the caller.

in-out parameter: These are to be initially allocated by the caller, then freed and re-allocated by the callee if necessary. As with out parameters, the caller is responsible for freeing the final returned value.

In the latter two cases, notice that is one piece of code that does the allocation, but another piece of code that does the freeing. In order for this to be successful, the two pieces of code must of course have knowledge of which memory allocator is being used. In OLE, it is often the case that the two pieces of code are written by independent development organizations. Thus, we must have rules about which memory allocators can be used when so that the system as a whole can function. The rules are as follows:

in parameter: either the task or the shared allocator may be used.

out parameter: only the task allocator may be used.

in-out parameter: only the task allocator may be used.

Further, the treatment of out and in-out parameters in failure conditions needs special attention. If a function returns a status code which is a failure code, then in general the caller has no way to clean up the out or in-out parameters returned to him. We therefore have the rules that:

out parameter: In error returns, out parameters must be reliably set to a value which will be cleaned up without any action on the caller's part. Further, it is the case that all out pointer parameters (usually passed in a pointer-to-pointer parameter, but which can also be passed as a member of a caller-allocate callee-fill struct) *must* explicitly be set to NULL. The most straightforward way to ensure this is to set these values to NULL on function entry.[39]

[39] This rule is stronger than it might seem to need to be in order to promote more robust application interoperability.

(On success returns, the semantics of the function of course determine the legal return values.)

in-out parameter: In error returns, all in-out parameters must either be left alone by the callee (and thus remaining at the value to which it was initialized by the caller; if the caller didn't initialize it, then it's an out parameter, not an in-out parameter) or be explicitly set as in the out parameter error return case.

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

4.5.1. IMalloc interface

IMalloc interface is used by the OLE libraries and by object handlers to allocate and free memory. An application provides an appropriate instance of IMalloc interface as a parameter to the OLE library initialization function CoInitialize(). The first three functions in this interface are merely a simple abstraction of the familiar C-library functions malloc(), realloc(), and free().

```
interface IMalloc : IUnknown {
    virtual     void *      Alloc(cb) = 0;
    virtual     void *      Realloc(pv, cb) = 0;
    virtual     void        Free(pv) = 0;
    virtual     ULONG       GetSize(pv) = 0;
    virtual     int         DidAlloc(pv) = 0;
    virtual     void        HeapMinimize() = 0;
    };
```

4.5.1.1. IMalloc::Alloc

void * IMalloc::Alloc(cb)

Allocate a memory block of at least cb bytes. The initial contents of the returned memory block are undefined. Specifically, it is not guaranteed that the block is zeroed. The block actually allocated may be larger than cb bytes because of space required for alignment and for maintenance information. If cb is 0, Alloc() allocates a zero-length item and returns a valid pointer to that item. This function returns NULL if there is insufficient memory available.

Always check the return from the this function, even if the amount of memory requested is small.

Argument	Type	Description
cb	ULONG	the number of bytes to allocate.
return value	void *	the allocated memory block, or NULL if insufficient memory exists.

4.5.1.2. IMalloc::Free

void IMalloc::Free(pv)

Deallocate a memory block. The pv argument points to a memory block previously allocated through a call to IMalloc::Alloc() or IMalloc::Realloc(). The number of bytes freed is the number of bytes with which the block was originally allocated (or reallocated, in the case of Realloc()). After the call, pv parameter is invalid, and can no longer be used. pv may be NULL, in which case this function is a no-op.

Argument	Type	Description
pv	void *	pointer to the block to free. May be NULL.

4.5.1.3. IMalloc::Realloc

void * IMalloc::Realloc(pv, cb)

Change the size of a previously allocated memory block. The pv argument points to the beginning of the memory block. If pv is NULL, Realloc() functions in the same way as IMalloc::Alloc() and allocates a new block of cb bytes. If pv is not NULL, it should be a pointer returned by a prior call to IMalloc::Alloc().

The cb argument gives the new size of the block in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block may be in a different location. Because the new block can be in a new memory location, the pointer returned by Realloc() is not guaranteed to be the pointer passed through the pv argument. If pv is not NULL and cb is 0, then the memory pointed to by pv is freed.

Realloc() returns a void pointer to the reallocated (and possibly moved) memory block. The return value is NULL if the size is zero and the buffer argument is not NULL, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

Argument	Type	Description
pv	void *	pointer to the block to reallocate. May be NULL.
cb	ULONG	the new size in bytes to allocate. May be zero.
return value	void *	the reallocated memory block, or NULL.

4.5.1.4. IMalloc::GetSize

ULONG IMalloc::GetSize(pv)

Return the size, in bytes, of the memory block allocated by a previous call to IMalloc::Alloc() or IMalloc::-Realloc() on this memory manager.

Argument	Type	Description
pv	void *	the pointer to be tested. May be NULL, in which case -1 is returned.
return value	ULONG	the size of the allocated memory block

4.5.1.5. IMalloc::DidAlloc

int IMalloc::DidAlloc(pv)

This function answers as whether or not the indicated memory pointer was allocated by the given allocator, if the allocator is able to determine that fact (many memory allocators will not be able to).

The values 1 (one) and 0 (zero) are returned as “did alloc” and “did not alloc” answers respectively; -1 (minus one) is returned if the IMalloc implementation is unable to determine whether it allocated the pointer or not. It is specifically permissible that the memory allocator passed to CoInitialize() *always* respond with -1 in this function.

Argument	Type	Description
pv	void *	the pointer to be tested. May be NULL, in which case -1 is returned.
return value	int	-1, 0, 1

4.5.1.6. IMalloc::HeapMinimize

void IMalloc::HeapMinimize()

Minimize the heap as much as possible by, for example, releasing unused memory in the heap to the operating system. This is useful in cases when a lot of allocations have been freed (using IMalloc::Free()) and the application wants to release the freed memory back to the OS so that it is available for other purposes.

4.5.2. Library initialization and memory management APIs

The following functions relate to initialization and memory management in the Component Object Model.

```
DWORD     CoBuildVersion();
HRESULT   CoInitialize(pMalloc);
```

```
void        CoUninitialize();
HRESULT     CoGetMalloc(dwMemContext, ppMalloc);
DWORD       CoGetCurrentProcess();
```

4.5.2.1. CoBuildVersion

DWORD CoBuildVersion()

Return the major and the minor version number of the Component Object Model library. Any given compiled version of an application can only run against one major version; it can use any of the minor versions. Thus, if the major version number reported by this function is different than that expected by the application, the application must *not* call CoInitialize(). Thus, this will always be the very first function that an application calls in the Component Object Model library.

Argument	Type	Description
return value	DWORD	a 32 bit value whose high-order 16 bits are the major version number and whose low-order 16 bits are the minor version number.

4.5.2.2. CoInitialize

HRESULT CoInitialize(pMalloc)

Initialize the Component Object Model library so that it can be used. With the exception of CoBuild-Version(), this function must be called by applications before any other function in the Component Object Model library.

Part of the design of the Component Object Model is that the application that owns a given process space (i.e.: the .EXE) has the opportunity to be in complete control of how memory used by components is allocated within that space. If it wishes to so be in control, then the application should construct an instance of the IMalloc interface and pass that instance to CoInitialize(); if the application does not wish to do anything special, then it should pass NULL to CoInitialize(), which will cause a default task-memory allocation implementation to be used.

Calls to CoInitialize() must be balanced by corresponding calls to CoUninitialize(). CoInitialize() needs to be called at least once per process that wishes to use the Component Object Model. Of course, only the initial CoInitialize() call in a given process will actually do the initialization, and only it's correspond-ingly balanced CoUninitialize() call will actually uninitialize. CoInitialize() will return `S_FALSE` if it has already been initialized in this process by a previous call.

Note that CoInitialize() is called internally by OleInitialize(); thus most applications will not in fact call CoInitialize() directly at all, but instead will simply call OleInitialize().

CoInitialize() follows the normal AddRef() / Release() rules described above with respect to the passed pMalloc pointer: since CoInitialize() wishes to retain the pMalloc pointer beyond the duration of the call, it calls pMalloc->AddRef().

Argument	Type	Description
pMalloc	IMalloc *	the memory allocator that is to be used for task memory by the OLE libraries and by object handlers. May be NULL, which will cause a default OLE-provided implementation to be used for the task allocator.
return value	HRESULT	`S_OK`, `S_FALSE`, `E_OUTOFMEMORY`

4.5.2.3. CoUninitialize

void CoUninitialize()

Shuts down the Component Object Model library, thus freeing any resources that it maintains. Among other things, it forces all RPC connections closed. Recall that CoInitialize() and CoUninitialize() calls need be balanced; thus, only the CoUninitialize() call that corresponds to the CoInitialize() call that actually did the initialization will actually uninitialize the library.

This function is called internally by OleUninitialize(); thus, most applications should not need to call CoUninitialize() directly.

4.5.2.4. CoGetMalloc

HRESULT CoGetMalloc(dwMemContext, ppMalloc)

This function retrieves from the OLE libraries either the task memory allocator originally passed to CoInitialize() or the OLE-provide shared memory allocator. Object handlers should use the task allocator returned by this function for their local memory management needs.

The particular allocator of interest is indicated by the dwMemContext parameter. Legal values for this parameter are taken from the enumeration MEMCTX:

```
typedef enum tagMEMCTX {
    MEMCTX_TASK = 1,
    MEMCTX_SHARED = 2,
    } MEMCTX;
```

MEMCTX_TASK returns the task allocator, the one passed to CoInitialize(). If CoInitialize() has not yet been called, NULL we be returned through ppMalloc and CO_E_NOTINITIALIZED is returned.

MEMCTX_SHARED returns the shared allocator. The shared allocator returned by this function is an OLE-provided implementation of IMalloc interface, one which allocates memory in such a way that it can be accessed by other applications. Further, memory allocated by this shared allocator in one application may be freed by the shared allocator in another. Except when a NULL pointer is passed, the shared memory allocator never answers -1 to IMalloc::DidAlloc(); it always indicates that either did or did not allocate the passed pointer. Pointers returned from the shared allocator *cannot* be directly manipulated using the host global memory allocator.

In versions of OLE2 on platforms other than Windows 16-bit or the Macintosh, it may perhaps be the case that the shared allocator may not be available. In such cases, CoGetMalloc(MEMCTX_SHARED, ...) will return E_INVALIDARG. People writing code now with future portability in mind will want to take this into account. Further, (advanced) people writing custom marshallers need to deal with the possible absence of shared memory no matter on which platform they are running; see the discussion of "destination contexts" in the write-up of remoting below.

Argument	Type	Description
dwMemContext	DWORD	a value from the enumeration MEMCTX.
ppMalloc	IMalloc **	the place in which the memory allocator should be returned.
return value	HRESULT	S_OK, CO_E_NOTINITIALIZED, E_INVALIDARG, E_OUTOFMEMORY

4.5.2.5. CoGetCurrentProcess

DWORD CoGetCurrentProcess()

Return a value unique to the current process. More precisely, return a value unique to the current process to the degree that it will not be reused until 2^{32} further processes have been created on the current work-station. One might be tempted to believe that the current HTASK available with GetCurrentTask() would work just as well as the value returned from this function; however, once a Windows task dies, its HTASK can be reused relatively quickly. If tables keyed by process must be maintained, then the value returned by CoGetCurrentProcess() offers significantly better robustness than the HTASK in the face of abnormal process termination.

Argument	Type	Description
return value	DWORD	a value unique to the current process

4.6. Class objects and IClassFactory interface

A central feature of OLE is that the code that knows how to manipulate a block of data can be opaquely located by a client and dynamically loaded. This is accomplished by associating with the data a tag known as a *class id*, a value of type CLSID. Given a CLSID, a client can load the associated executable code by calling CoGetClassObject(), to which it passes the interface by which it wishes to communicate with the class. Most often, this is the interface IClassFactory, through which it can make an instance of the class, though other interfaces are appropriate in some rare situations. When calling IClassFactory::Create-Instance(), the client passes as parameter the interface by which it wishes to talk to the resulting object; it is with this interface that the client then initializes the newly-created object. For OLE compound-document (embedded and linked) objects, this whole process is conveniently encapsulated by the function OleLoad(), and as a result applications usually need not understand the process in detail.

Server and handler implementors implement one class object (i.e.: one instance of IClassFactory) for each class that they support.[40] Server implementors register the public availability of their class objects with CoRegisterClassObject() in order that they may be connected to by OLE container applications.

The following interfaces and functions relate directly to creating instances of classes:

```
interface IClassFactory[41] : IUnknown {
    virtual    HRESULT    CreateInstance(punkOuter, iid, ppvObject) = 0;
    virtual    HRESULT    LockServer(fLock) = 0;
    };

typedef ... CLSID;                    // a tag used for dynamically locating & loading implementations

HRESULT    CoGetClassObject(clsid, grfContext, pvReserved, iid, ppv);
HRESULT    CoRegisterClassObject(rclsid, pUnk, grfContext, grfFlags, pdwRegister);
HRESULT    CoRevokeClassObject(dwRegister);
HRESULT    DllGetClassObject(clsid, iid, ppv);        // function exported from dlls
```

The following are somewhat lower level functions which manage the loading and freeing of DLL-based class implementations. Most programmers will not have need to call these directly.

```
HINSTANCE  CoLoadLibrary(lpszLibName, fAutoFree);
void       CoFreeUnusedLibraries();
void       CoFreeLibrary(hInst);
void       CoFreeAllLibraries();
HRESULT    DllCanUnloadNow();                  // function exported from dlls
```

4.6.1. How to Create an Instance of a Class

Creating an instance of a class is accomplished using the appropriate variation on the following steps.

1. What class do you want to instantiate?

This step identifies the CLSID of which an instance is desired. Depending on the task at hand, this is done very differently from situation to situation. The following are some examples:

- In the Insert Object Dialog, each type of object in the list has a unique associated CLSID. If a user wants to insert a given type, then that is the CLSID of which an instance is needed.
- When a previously-saved object is loaded from persistent storage, the class of object which should be used to manipulate said object is identified with a CLSID which is kept in persistent storage along with the object's data.

---

[40] In this respect, IClassFactory takes the place of the interface OLESERVER found in OLE 1.

[41] Note: "IClassFactory" would perhaps more appropriately be named "IObjectFactory" since using it one creates objects, not classes. But, alas, it is not so named.

2. Get your hands on the class factory for that class.

This is a call to CoGetClassObject() (note that this code and the other examples in this section omit error checking in the interests of exposition):

```
IClassFactory* pcf;
CoGetClassObject(clsid, CLSCTX_INPROC, 0, IID_IClassFactory, (void**)&pcf);
```

In different instance creation situations, the immediately preceding line usually varies only in the `CLSCTX` value or values used.

3. Create an uninitialized instance of the class.

An uninitialized instance is created using IClassFactory::CreateInstance(). The interface requested from the object should be the interface by which the object is initialized. For OLE embeddings, this interface is IPersistStorage, but the particular interface used depends heavily on the situation in question. If creating the object as part of an aggregate, the Controlling Unknown is also passed.

```
IPersistStorage* pPersistStorage;
pcf->CreateInstance(NULL, IID_IPersistStorage, (void**)&pPersistStorage);          // non-aggregate case
pcf->Release();
```

4. Initialize the newly created instance

This involves invoking one or more methods in the requested initialization interface, and as such, is completely dependent on the situation. For OLE embeddings, a new blank object is initialized with IPersistStorage::InitNew(), while a reloaded object is initialized with IPersistStorage::Load().

```
pPersistStorage->InitNew();
```

5. Query for some working interface on the instance and clean up

Often, the interface needed on an object once it has been initialized is different than the interface by which initialization is done. In this final step, we query the object for the working interface we need, then clean up by releasing previously obtained pointers:

```
IOleObject * pOleObject;
pPersistStorage->QueryInterface(IID_IOleObject, (void**)&pOleObject);
pPersistStorage->Release();
```

The following helper function encapsulates the common cases of this paradigm.

4.6.1.1. CoCreateInstance

HRESULT CoCreateInstance(rclsid, pUnkOuter, grfContext, iid, ppv)

Create an instance of the indicated class, asking for initialization interface iid. In short, this little helper function performs Step 2. and Step 3. listed above.

Argument	Type	Description
rclsid	REFCLSID	the class of which an instance is desired
pUnkOuter	IUnknown*	the Controlling Unknown, if any.
grfContext	DWORD	the CLSCTX to be used.
iid	REFIID	the initialization interface desired
ppv	void**	the place at which the initialization interface is to be returned.
return value	HRESULT	S_OK, any error that can be returned by either CoGetClassObject() or IClassFactory::CreateInstance()

4.6.2. Creating New CLSIDs & IIDs

We have seen how classes are identified by CLSIDs. A natural question for a class implementor to ask is "How do I generate a CLSID for my class?" Due to OLE2 implementation considerations, the answer to this question varies according to whether a short-term or a long-term answer is being sought.

In the short term, CLSIDs are generated with the help of the tool UUIDGEN.EXE, provided as part of the OLE 2 software development kit. This is a DOS tool that when run writes to standard output the string representation of a new CLSID. The provided implementation of UUIDGEN.EXE requires that a network card be installed on the machine. Simply put, UUIDGEN.EXE uses the machine id obtained from this card together with the current date and time to come up with a new unique class identifier. An example of the use of this tool is the following.

```
[C:\WIN] uuidgen
80C11F40-7503-1068-8576-00DD01113F11
```

The string representation of a CLSID, as determined by StringFromCLSID(), is in fact the string generated by UUIDGEN.EXE surrounded by two braces:

```
CLSID clsid;
CLSIDFromString( "{80C11F40-7503-1068-8576-00DD01113F11}" , &clsid);
```

In the long term, it is presently expected that two things will be done:

- UUIDGEN.EXE will be enhanced so as not to require the presence of a network card.
- An API function will be provided which returns a new CLSID directly.

Interface identifiers (IIDs) are allocated in the almost exactly the same way. Merely use IIDFromString() instead of CLSIDFromString().

See also the macro DEFINE_GUID in compobj.h and initguid.h.

```
DEFINE_GUID(IID_NewInterface, 0x80C11F40, 0x7503, 0x1068, 0x85, 0x76, 0x00, 0xdd, 0x01, 0x11, 0x3f, 0x11)
```

CLSIDs and IIDs can also be allocated by contacting Microsoft.

4.6.3. Class & Instance API & Interface Details

The remainder of this section described the interfaces and functions relating to class management and instance creation.

4.6.3.1. CoGetClassObject

HRESULT CoGetClassObject(clsid, grfContext, pvReserved, iid, ppv)

Locate and connect to the class object associated with the class tag clsid. If necessary, executable code will be dynamically loaded in order to accomplish this. The interface by which the caller wishes to talk to the class object is indicated by iid; this is almost always IID_IClassFactory.

Different pieces of code can be associated with one class tag for use in different execution contexts. The context in which the caller is interested is indicated by the parameter grfContext, a group of flags taken from the enumeration CLSCTX:

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

The several contexts are tried in the sequence in which they are listed here. Multiple values may be or'd together indicating that multiple contexts are acceptable to the caller:

```
#define CLSCTX_INPROC       (CLSCTX_INPROC_SERVER|CLSCTX_INPROC_HANDLER)
#define CLSCTX_SERVER       (CLSCTX_INPROC_SERVER|CLSCTX_LOCAL_SERVER)
```

These context values have the following meanings.

Value	Description
CLSCTX_INPROC_SERVER	Load the in-client-process code which creates and manages the objects of this class.
CLSCTX_INPROC_HANDLER	Load the in-client-process code (a DLL) which implements client-side structures of this class when instances of it are accessed remotely.
CLSCTX_LOCAL_SERVER	Load the separate-process code (an EXE) which creates and manages the objects of this class. The EXE is launched by the system with "-Embedding"[42] on its command line.[43]

The implementation of this function operates by consulting as appropriate for the context indicated both the registration database and the extant class objects published with CoRegisterClassObject().

Some sample uses of particular combinations of these flags are as follows:

Function	Flags
OleLoad()	CLSCTX_INPROC_HANDLER | CLSCTX_INPROC_SERVER Putting an OLE object into the loaded state requires in-process access, but we don't care if we get all the object functionality at the moment or not.
OleRun()	CLSCTX_LOCAL_SERVER Running an OLE object requires that we connect to the full code of the object, wherever it might be.
CoUnmarshalInterface()	CLSCTX_INPROC_HANDLER Unmarshalling, by its nature, needs the form of the class designed for remote access.

The arguments to this function are as follows:

Argument	Type	Description
clsid	REFCLSID	that class that is to be loaded.
grfContext	DWORD	the context in which the executable code is to run.
pvReserved	void *	reserved for future use. Must be NULL.
iid	REFIID	the interface with which we want to talk to the class object.
ppv	void **	the place in which to put the resultant interface.
return value	HRESULT	S_OK, REGDB_E_CLASSNOTREG, E_OUTOFMEMORY

4.6.3.2. IClassFactory::CreateInstance

HRESULT IClassFactory::CreateInstance(punkOuter, iid, ppvObject)

Create an uninitialized instance of this class. Initialization is done subsequently using some interface-specific function. Examples of common initialization functions include IPersistStorage::InitNew(), IPersistStorage::Load(), IPersistStream::Load(), and IPersistFile::Load().

punkOuter indicates if the object is being created as part of an aggregation. Class implementations need to be consciously designed to be able to be so used, and not all classes are so designed.

---

[42] case insensitive. "/Embedding" should also be treated as synonymous.

[43] Incidentally, OLE1 applications also add "-Embedding" to the command line when running servers, but applications need not take any special action to deal with this scenario, as the interaction is transparently handled by the compatibility layer. However, when connecting links, OLE1 applications launch servers with "-Embedding *filename*" on the command line. OLE2 servers *do need* to check for the presence of the filename. If there, the OLE2 application must load the file and register that fact in the Running Object Table. In addition, if the OLE2 application is a *single use* server the application should *not* register its class factory, since (as the application is single-use) it is already being used. Further, it must do this all before yielding; such is OLE1. If you think about it, this special behaviour does essentially the same thing as pretending that an instance was created, IPersistFile was requested, and Load() used to open file.

Argument	Type	Description
punkOuter	IUnknown *	may be NULL, as is very often the case. If non-NULL, then the object is being created as part of an aggregate, and this is the controlling unknown of the aggregation as a part of which the new object is being instantiated. See the discussion of aggregation in the previous chapter.
iid	REFIID	the interface by which the caller wishes to talk to the resulting object and through which it initializes the object.
ppvObject	void **	the place in which the resulting object is to be returned.
return value	HRESULT	S_OK, CLASS_E_NOAGGREGATION, E_OUTOFMEMORY

4.6.3.3. IClassFactory::LockServer

HRESULT IClassFactory::LockServer(fLock)

This function can be called by client of a class factory in order to keep an open server application in memory. Keeping the server application in memory allows instances of this class to be created more quickly than they otherwise would be. In the absence of locking, local servers can CoDisconnectObject() themselves and shut down.

Most clients have no need to call this function. It is present primarily for the benefit of sophisticated clients with special performance needs from certain classes.

It is an error to call IClassFactory::LockServer(TRUE) and then Release() the IClassFactory without first releasing the lock with IClassFactory::LockServer(FALSE). The reasoning goes that he who locks the server is responsible for unlocking it, and once the class factory is released, there is no mechanism by which the caller can be guaranteed to later connect to the same class object (consider single use classes, for example). See also CoRegisterClassObject() below.

Argument	Type	Description
fLock	BOOL	true if a lock is being added to the class factory, false if one is being removed.
return value	HRESULT	S_OK, E_FAIL.

4.6.3.4. CoRegisterClassObject

HRESULT CoRegisterClassObject(rclsid, pUnk, grfContext, grfFlags, pdwRegister)

The CoRegisterClassObject() function registers the specified server class object with the OLE library in order that it may be connected to by other applications. When a server application starts, it creates an IClassFactory instance and calls this function. Servers that support several different classes allocate a different class factory instance for each. When a server application exists, it revokes all its registered class objects with CoRevokeClassObject().

This function is only called by "local servers;" it should not be called by "in-proc servers" or by "in-proc handlers." See DllGetClassObject().

A server application for a class object should revoke (using CoRevokeClassObject()) the registration of said class object made by CoRegisterClassObject() when

- there are no instances of the class in existence, and
- the class object has a zero number of locks, and
- the application servicing the class object is not showing itself to the user (that is, is not under user control)

When, subsequently, the reference count on the class object reaches zero, the class object can be destroyed, allowing the application to exit.

Thus, a sophisticated client of a class can keep a server executable alive by obtaining its class object and calling IClassFactory::Lock(TRUE) on it. Note though, that since some class objects are single use, the

sophisticated client must be sure to hold on to the *actual class object* from which he instantiates objects. This prohibits said sophisticated client, for example, from using helper functions like OleLoad(); instead, it will have to code variants of such functions itself.

The grfFlags is used to control how connections are made to the class object. Values for this parameter are the following:

```
typedef enum tagREGCLS {
    REGCLS_SINGLEUSE = 0,
    REGCLS_MULTIPLEUSE = 1,
    } REGCLS;
```

Value	Description
REGCLS_SINGLEUSE	once one client has connected to the class object with CoGetClassObject(), then the class object should be removed from public view so that no other clients can similarly connect to it. This flag will commonly be given for single document interface (SDI) applications. Specifying this flag does not affect the responsibility of the server to call CoRevokeClassObject(); this must always be done.
REGCLS_MULTIPLEUSE	Many CoGetClassObject() calls can connect to the same class object. (In-proc servers and handlers are by their nature always multiple use, though they do not call CoRegisterClassObject()).

The arguments to this function are as follows:

Argument	Type	Description
rclsid	REFCLSID	the id of the class being registered.
pUnk	IUnknown *	the class object whose availability is being published.
grfContext	DWORD	as in CoGetClassObject().
grfFlags	DWORD	REGCLS values that control the use of the class object.
pdwRegister	DWORD *	a place at which a token is passed back with which this registration can be revoked in CoRevokeClassObject().
return value	HRESULT	S_OK, CO_E_OBJISREG, E_OUTOFMEMORY

4.6.3.5. CoRevokeClassObject

HRESULT CoRevokeClassObject(dwRegister)

The CoRevokeClassObject() function is called by a server application to inform the libraries that a class object previously registered with CoRegisterClassObject() is no longer available for use.

See CoRegisterClassObject() for a discussion as to when CoRevokeClassObject() should be called.

Argument	Type	Description
dwRegister	DWORD	a token previously returned from CoRegisterclassObject().
return value	HRESULT	S_OK, E_INVALIDARG

4.6.3.6. CoLoadLibrary

HINSTANCE CoLoadLibrary(lpszLibName, fAutoFree)

Load the library (i.e.: the DLL) of the given name into the caller's process. Normally, this is not called directly by clients, but is instead called internally if needed by CoGetClassObject(). Internally, loaded libraries are reference counted, with CoLoadLibrary() incrementing the count, and CoFreeLibrary() decrementing it.

Argument	Type	Description
lpszLibName	LPSTR	the name of the library that is to be loaded. The use of this name is the same as in the function LoadLibrary().

fAutoFree	BOOL	if true, then this library will be freed whenever it is no longer needed, either through CoFreeUnusedLibraries(), or at uninitialization time, in CoUninitialize(). If false, then the library should be explicitly freed with CoFreeLibrary().
return value	HINSTANCE	the module handle of the now-loaded library. NULL indicates failure.

4.6.3.7. CoFreeUnusedLibraries

void CoFreeUnusedLibraries()

This function and unloads any DLLs that have been loaded as a result of previous CoLoadLibrary(..., true) calls but which are no longer in use. Applications can call this function periodically to free up resources, either at the top of their message loop or in some idle-time task.

4.6.3.8. CoFreeLibrary

void CoFreeLibrary(hInst)

Free a library that was previously loaded with CoLoadLibrary(..., false). That is, it is illegal to explicitly free a library that whose corresponding CoLoadLibrary() call specified auto-free.

Argument	Type	Description
hInst	HINSTANCE	the module handle which is to be freed.

4.6.3.9. CoFreeAllLibraries

void CoFreeAllLibraries()

Free all the libraries that have been loaded by the calling process as a result of CoLoadLibrary() calls, irrespective of whether they are currently in use or not, and irrespective of the fAutoFree flag by which they were loaded (though the debug version of the system prints a warning if there are any outstanding non-auto-free loads).

This function is called internally by CoUninitialize(); thus applications normally have no need to call this function directly.

4.6.3.10. DllGetClassObject

HRESULT DllGetClassObject(clsid, iid, ppv)

This is not a function in the OLE-provided libraries; rather, it is a function that is exported from DLLs supporting the Component Object Model.

In the case that a call to the OLE API function CoGetClassObject() results in the class object having to be loaded from a DLL, CoGetClassObject() uses the DllGetClassObject() that must be exported from the DLL in order to actually retrieve the class.

Argument	Type	Description
clsid	REFCLSID	that class that is to be loaded.
iid	REFIID	the interface with which the caller wants to talk to the class object. Most often this is IID_IClassFactory.
ppv	void **	the place in which to put the resultant interface.
return value	HRESULT	S_OK, E_NOMEMORY

4.6.3.11. DllCanUnloadNow

HRESULT DllCanUnloadNow()

This is not a function in the OLE-provided libraries; rather, it is a function that is exported from DLLs supporting the Component Object Model.

This function, which should be exported from DLLs designed to be dynamically loaded in CoGetClassObject(), answers whether the DLL is no longer in use and should be unloaded. In short, such a DLL will no longer be in use when there are no extant instances of classes that it manages. If a DLL loaded by CoGetClassObject() fails to export this function, then the DLL will be forcibly unloaded in CoUninitialize().

Argument	Type	Description
return value	HRESULT	S_OK if the DLL should be unloaded, S_FALSE otherwise.

4.7. Interface Remoting: Remote Procedure Calling and Marshalling

In the Component Object Model, clients communicate with objects solely through the use of vtable-based interface instances. The state of the object is manipulated by invoking functions on those interfaces. For each interface method, the object provides an implementation which does the appropriate manipulation of the object internals.

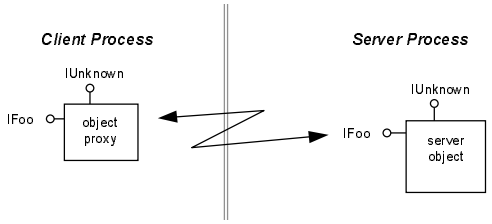
The underlying goal of interface remoting is to provide infrastructure and mechanisms such that the client and the server objects can in fact be in different processes. Thus, when the client makes a call on an interface of the object, a process transition must be made to the server process, the work carried out, and a return process transition made back to the client process.

A significant subgoal is that this infrastructure be transparent: it must not be the case that either client or object is necessarily aware that the other party is in fact in a different process; the glue that makes this happen must be automatically stuck in the loop at the right time.

The crux of the problem to be addressed in interface remoting can thus be summarized as follows:

> "Given an already existing remoted-interface connection between a client process and a server process, how can a method invocation through that connection return a new interface pointer so as to create a second remoted-interface connection between the two processes?"

We state the problem in this way so as to avoid for the moment the issue of how an initial connection is made between the client and the server process; we will return to that later.

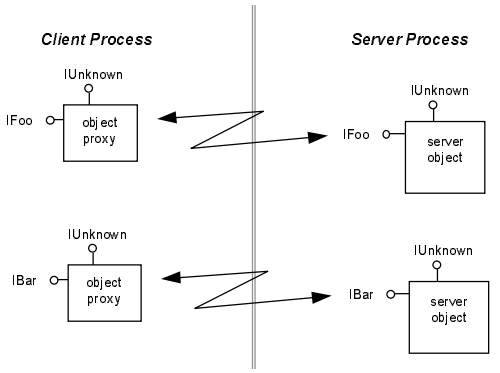


Let's look at an example. Suppose we have an object in a server process which supports an interface IFoo, and that interface of the object (and IUnknown) has sometime in the past been remoted to a client process through some means not here specified. In the client process, there is an object proxy which supports the exact same interfaces as does the original server object, but whose *implementations* of methods in those interfaces are special, in that they forward calls they receive on to calls on the real method implementations back in the server object. We say that the method implementations in the object proxy *marshal* the data, which is then conveyed to the server process, where it is *unmarshalled.* That is, "marshalling" refers to the packaging up of method arguments for transmission to a remote process; "unmarshalling" refers to the unpackaging of this data at the receiving end. Notice that in a given call, the method arguments are marshalled and unmarshalled in one direction, while the return values are marshalled and unmarshalled in the other direction.

For concreteness, let us suppose that the IFoo interface is defined as follows:

```
interface IFoo : IUnknown {
    virtual     IBar *      ReturnABar() = 0;
    };
```


If the in the client process pFoo->ReturnABar() is invoked, then the object proxy will forward this call on to the IFoo::ReturnABar() method in the server object, which will do whatever this method is supposed to do in order to come up with some appropriate IBar*. The server object is then required to return this IBar* back to the client process. The act of doing this will end up creating a second connection between the two processes:



It is the procedure by which this second connection is established which is the subject of our discussion here. This process involves two steps:

1. On the server side, the IBar* is packaged or marshalled into a data packet.
2. The data packet is conveyed by some means to the client process, where the data it contains is unmarshalled to create the new object proxy.

The process begins with the code doing the marshalling of the returned IBar*. This code has in hand a pointer to an interface that it knows in fact to be an IBar*. The first step in marshalling involves finding out whether the object of which this is an interface in fact supports *Custom Marshalling*. Custom marshalling is a mechanism that permits an object to be in control of creation of remote object proxies to itself. In certain situations, Custom Marshalling can be used to create a more efficient object proxy than would otherwise be the case. Use of Custom Marshalling is completely optional on the object's part; if the object chooses not to support Custom Marshalling, then *Standard Marshalling* is used to marshal the IBar*. Standard marshalling uses a system-provided object proxy implementation in the client process. This standard object proxy is a generic piece of code; it can be used as the object proxy for any interface on any object. However, the act of marshalling (and unmarshalling) method arguments and return values is inherently interface-specific, since it is highly sensitive to the semantics of the particular methods in question. To accommodate this, the standard object proxy dynamically loads in interface-specific pieces of code as needed in order to do the marshalling.

Let's examine how Custom Marshalling works.

4.7.1. Architecture of Custom Marshalling

Imagine that we are presently in a piece of code whose job it is to marshal an interface pointer that it has in hand. For clarity, in what follows we'll refer to this piece of code as the "original marshalling stub." The general case is that the original marshalling stub does not *statically*[44] know the particular interface identifier (IID) to which the pointer conforms; the IID may be passed to this code as a second parameter. This is a common paradigm in OLE2. Examples include:

```
IUnknown::QueryInterface(REFIID riid, void** ppvObject);
IOleItemContainer::GetObject(..., REFIID riid, void** ppvObject);
IClassFactory::CreateInstance(..., REFIID riid, void** ppvNewlyCreatedObject);
```

For the moment, let us assume the slightly less general case where the marshalling stub in fact does know a little bit about the IID: in particular, let us assume that it knows that the interface in fact derives from IUnknown (we'll discuss later the situation in which this is not true).

To find out whether the object to which it has an interface supports Custom Marshalling, the original marshalling stub simply does a QueryInterface() for the interface IMarshal. That is, an object signifies that it wishes to do Custom Marshalling simply by implementing the IMarshal interface. IMarshal is defined as follows:

```
interface IMarshal : IUnknown {
    virtual    HRESULT    GetUnmarshalClass(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pclsid) = 0;
    virtual    HRESULT    GetMarshalSizeMax(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pcb) = 0;
    virtual    HRESULT    MarshalInterface(pstm, iid, pvInterface, dwDestContext, pvDestContext, mshlflags) = 0;
    virtual    HRESULT    UnmarshalInterface(pstm, iid, ppvInterface) = 0;
```

[44] i.e.: at compile time of the original marshalling stub

```
virtual     HRESULT     DisconnectObject(dwReserved) = 0;
virtual     HRESULT     ReleaseMarshalData(pstm) = 0;
};
```

The idea is that if the object says "Yes, I do want to do Custom Marshalling" that the original marshalling stub will use this interface in order to carry out the task. The sequence of steps that carry this out is:

1. Using GetUnmarshalClass(), the original marshalling stub asks the object which kind of (i.e.: which class of) proxy object it would like to have created on its behalf in the client process.
2. (optional) Using GetMarshalSizeMax(), the stub asks the object how big of a marshalling packet it will need. The object will return an upper bound on the amount of space it will need.
3. The marshalling stub allocates a marshalling packet of appropriate size, then creates an IStream* which points into the buffer. Unless in the previous step the object gave an upper bound on the space needed, the IStream* must be able to grow its underlying buffer dynamically as IStream::Write() calls are made.
4. The original marshalling stub asks the object to marshal its data using MarshalInterface().

We will discuss the methods of this interface in detail in a moment.

At this point, the contents of the memory buffer pointed to by the IStream* together with the class tag returned in step (1) comprises all the information necessary in order to be able to create the proxy object in the client process. It is the nature of remoting and marshalling that "original marshalling stubs" such as we have been discussing know how to communicate with the client process; recall that we are assuming that an initial connection between the two processes had already been established. The marshalling stub now communicates to the client process, by whatever means is appropriate, the class tag and the contents of the memory that contains the marshalled interface pointer. In the client process, the proxy object is created as an instance of the indicated class using the standard Component Object Model instance creation paradigm. IMarshal is used as the initialization interface; the initialization method is IMarshal::UnmarshalInterface(). The unmarshalling process looks something like the following:

```
void ExampleUnmarshal(CLSID& clsidProxyObject, IStream* pstm, IID& iidOriginallyMarshalled, void** ppvReturn)
{
    IClassFactory* pcf;
    IMarshal* pmsh;
    CoGetClassObject(clsidProxyObject, CLSCTX_INPROC_HANDLER,  NULL, IID_IClassFactory, (void**)&pcf);
    pcf->CreateInstance(NULL, IID_IMarshal, (void**)pmsh);
    pmsh->UnmarshalInterface(pstm, iidOriginallyMarshalled, ppvReturn);
    pmsh->Release();
    pcf->Release();
}
```

There are three important reasons why an object may choose to do Custom Marshalling. First, objects which already are proxy objects can use Custom Marshalling to avoid creating proxies to proxies; new proxies are instead short-circuited back to the original server. This is both an important efficiency and an important robustness consideration. Second, object implementations whose whole state is kept in shared memory can often be remoted by creating an object in the client that talks directly to the shared memory rather than back to the original object. This can be a significant performance improvement, since access to the remoted object does not result in context switches. The Compound File implementations of IStorage and IStream are important examples of this use of Custom Marshalling. Third, some objects are of the nature that once they have been created, they are immutable: their internal state does not subsequently change. Many monikers are an example of such objects. These sorts of objects can be efficiently remoted by making independent copies of themselves in client processes. Custom marshalling is the mechanism by which they can do that, yet have no other party be the wiser for it.

However, Custom Marshalling may not be used by OLE2 embeddings; it is intended primarily for other situations such as Monikers, Compound Files, etc. More correctly, Custom Marshalling may not be used by OLE2 embeddings which are not completely implemented in an INPROC_SERVER. This restriction arises because of the fact architecture of how an OLE2 embedding handler, created when the object enters

the loaded state, communicates with the Local Server as the running state is entered. It is possible that this restriction may be removed in the future.

4.7.2. Architecture of Standard Marshalling

If the object being remoted does not support Custom Marshalling, signified by the lack of support for IMarshal interface, then Standard Marshalling is used instead. With Standard Marshalling, the actual marshalling and unmarshalling of interface function parameters is handled by the system. However, the object being marshalled is given a second chance to indicate that it would like code that it specifies to run in the client process. Such code would presumably handle some processing locally, but refer the majority of requests back to the original object using the system supplied mechanism.

This is accomplished in the following way. Once the system has been determined that Standard Marshalling is to be used, the object is queried in order for IStdMarshalInfo and IPersist. If either of these interfaces is supported, then the CLSID returned by invoking the one method contained in each is used to identify the handler that is to be loaded in the client context (see CoGetClassObject()). The handler of this class must use the standard remoting connection architecture. Presently, this means that such handlers must aggregate in the OLE2 Default Handler, as is described in OleCreateDefaultHandler().[45]

If neither of these interfaces is supported, than a vanilla handler which merely remotes all calls back to the original object is used. For components which are not embeddings, this is likely to be the common situation. It corresponds to the classic RPC scenario, where the remote proxy is little more than a forwarder of requests.

4.7.2.1. IStdMarshalInfo::GetClassForHandler

HRESULT IStdMarshalInfo::GetClassForHandler(dwDestContext, pvDestContext, pClsid)

Retrieves the class identifier used to determine the handler in the destination process that is used in standard marshaling.

Server applications which support class conversion (**Activate As** in the **Convert** dialog box) must implement the IStdMarshalInfo interface. Implementation is necessary for the correct handler to be determined in all cases. See also the discussion of **Activate As** in the chapter on "Persistent Storage for Objects."

Argument	Type	Description
dwDestContext	DWORD	The type of destination context to which this object is being passed.
pvDestContext	void *	Pointer to the destination context.
pClsid	CLSID*	A pointer to where to return the handler's class identifier.
return value	HRESULT	S_OK

4.7.3. Storing Marshalled Interface Pointers in Global Tables

In normal marshalling usage, interface pointers which are marshalled are merely transported across the "wire" to the other side (the other process), where they are unmarshalled. In this usage, the data packet that results from the marshalling process is unmarshalled exactly once. In contrast, there are occasions where we have need to marshal an interface pointer and store it in a globally accessible table. Once in the table, the data packet can be retrieved and unmarshalled zero, one, or more times. The Running Object Table and the table maintained by CoRegisterClassObject() are examples of this situation. In effect, the marshalled data packet sitting in the table acts very much like another pointer to the object. Depending on the semantics of the table in question, the "data packet pointer" may need to either act as a reference-counted or non-reference-counted pointer to the interface. That is, depending on in which table the object is placed, the presence of the object in the table either does or does not keep the object alive. Further, because of this behaviour, we must be careful to have marshalling-specific code execute at the time that

[45] It is presently intended that future OLE releases will expose the standard remoting connection architecture independently of the Default Handler, but such is not the case at this time.

these data-packets are removed from these tables and destroyed. We cannot simply throw the packets away, as the presence or absence of the internal state that they maintain may be important to the object that they indicate.

Technically, we address this space of possibilities in the following way. When an interface pointer is marshalled it is told by a parameter for which of the following three reasons it is being marshalled.

1. This is a normal marshal-then-unmarshal-once case.
2. This is a marshal for storing into a global table case, and the presence of the entry in to the table *is* to count as an additional reference to the interface.
3. This is a marshal for storing into a global table case, and the presence of the entry in to the table *is not* to count as an additional reference to the interface.

Further, whenever, a Case 2) or Case 3) marshalled-data-packet is removed from the table, it is the responsibility of the table implementor to call CoReleaseMarshalData().

4.7.4. Creating an Initial Connection Between Processes

Earlier we said we would later discuss how an initial remoting connection is established between two processes. It is now time to have that discussion.

The real truth of the matter is that the initial connection is established by some means outside of the architecture that we have been discussing here. The minimum that is required is some primitive communication channel between the two processes. As such, we cannot hope to discuss all the possibilities. But we will point out some common ones.

One common approach, used heavily in OLE 2.0, is that initial connections are established just like other connections: an interface pointer is marshalled in the server process, the marshalled data packet is ferried the client process, and it is unmarshalled. The only twist is that the ferrying is done by some means *other* than the RPC mechanism which we've been describing. There are many ways this could be accomplished. Among them are:

- The server process could put the packet in some globally memory table, from which the client process could retrieve it. In OLE 2.0, examples of this technique include the Running Object Table and the running server table used internally by CoGetClassObject()/CoRegisterClassObject().
- The server process could put the data in a file, from which the client process could receive it.
- (:-) The server process could print a block of data on the screen, then have the user type the data into the client process.

You get the idea...

Another common approach likely to be prevalent in networking situations is that a centralized directory service is used.

4.7.5. Remoting-Related Function Descriptions

The following functions are related to interface remoting:

```
HRESULT     CoMarshalInterface(pstm, riid, pUnk, dwDestContext, pvDestContext, mshlflags);
HRESULT     CoUnmarshalInterface(pstm, iid, ppv);
HRESULT     CoDisconnectObject(pUnkInterface, dwReserved);
HRESULT     CoReleaseMarshalData(pstm);
HRESULT     CoGetStandardMarshal(iid, pUnkObject, dwDestContext, pvDestContext, mshlflags, ppmarshal);

HRESULT     CoMarshalHresult(pstm, hresult);
HRESULT     CoUnmarshalHresult(pstm, phresult);

HRESULT     CoLockObjectExternal(pUnk, fLock, fLastUnlockReleases);

typedef enum tagMSHLFLAGS {
    MSHLFLAGS_NORMAL = 0,
    MSHLFLAGS_TABLESTRONG = 1,
```

```
    MSHLFLAGS_TABLEWEAK = 2,
    } MSHLFLAGS;
```

4.7.5.1. CoMarshalInterface

HRESULT CoMarshalInterface(pstm, riid, pUnk, dwDestContext, pvDestContext, mshlflags)

Marshal the interface riid on the object on which pUnk is an IUnknown* into the given stream in such a way as it can be reconstituted in the destination using CoUnmarshalInterface().[46] This the root level function by which an interface pointer can be marshalled into a stream. It carries out the test for Custom Marshalling, using it if present, and carries out Standard Marshalling if not. This function is normally only called by code in interface proxies or interface stubs that wish to marshal an interface pointer parameter, though it will sometimes also be called by objects which support Custom Marshalling.

This function is, in fact, a helper function in that it carries out nothing internally that is not otherwise publicly available.[47]

riid indicates the interface on the object which is to be marshalled. It is specifically *not* the case that pUnk need actually be of interface riid; this function will QueryInterface from pUnk to determine the actual interface pointer to be marshalled.

However, in OLE 2.0 the implementation of this function is limited: presently, pUnk must indeed *be of interface riid. Rather than QueryInterface()ing, the present implementation merely does a cast. Thus, until this is remedied in a subsequent OLE release, the caller of this function is responsible for ensuring that the pointer passed to pUnk is in fact of interface riid. As a consequence, at the moment only interfaces derived from IUnknown can be marshalled.*

dwDestContext identifies the execution context relative to the current context in which the unmarshalling will be done. Different marshalling might be done, for example, depending on whether the unmarshal happens on the same workstation vs. on a different workstation on the network; an object could choose to do Custom Marshalling in one case but not the other. The legal values for dwDestContext are taken from the enumeration MSHCTX, which presently contains the following values.

```
tyepdef enum tagMSHCTX {
    MSHCTX_NOSHAREDMEM = 1,    // only lower most bit is significant
    } MSHCTX;
```

These flags have the following meanings.

Value	Description
MSHCTX_NOSHAREDMEM	The unmarshalling context does not have shared memory access with the marshalling context.

In the future, more MSHCTX flags will be defined, particularly when network-remoting is implemented. pvDestContext is a parameter reserved for the use of future-defined MSHCTX's. ppvDestContext parameters may not be stored in the internal state of custom marshallers.

mslflags indicates the purpose for which the marshal is taking place, as was overviewed in an earlier part of this document. Values for this parameter are taken from the enumeration MSHLFLAGS, and have the following interpretation.

Value	Description
MSHLFLAGS_NORMAL	The marshalling is occurring because of the normal case of passing an interface from one process to another. The marshalled-data-packet that results from the call will be transported to the other process, where it will be unmarshalled (see CoUnmarshalInterface()).

[46] That is, the mechanism for unmarshalling a marshalled interface pointer is the *same* irrespective of whether the marshalling was done using custom or standard marshalling.

[47] *Due to the limitations of CoGetStandardMarshal() in OLE 2.0, CoMarshalInterface is in fact* not *a helper function, since internally it contains the only mechanism by which the standard marshalling mechanism can be invoked. However, when CoGetStandard-Marshal() is fully implemented (in a later OLE release), CoMarshalInterface() will indeed be a helper function. At that time, we will document what it does internally.*

	With this flag, the marshalled data packet will be unmarshalled either one or zero times. CoReleaseMarshalData() is always (eventually) called to free the data packet.
MSHLFLAGS_TABLESTRONG	The marshalling is occurring because the data-packet is to be stored in a globally-accessible table from which it is to be unmarshalled zero, one, or more times. Further, the presence of the data-packet in the table is to count as a reference on the marshalled interface. When removed from the table, it is the responsibility of the table implementor to call CoReleaseMarshalData() on the data-packet.
MSHLFLAGS_TABLEWEAK	The marshalling is occurring because the data-packet is to be stored in a globally-accessible table from which it is to be unmarshalled zero, one, or more times. However, the presence of the data-packet in the table is *not* to count as a reference on the marshalled interface. Destruction of the data-packet is as in the MSHLFLAGS_TABLESTRONG case.

A consequence of this design is that the marshalled data packet will want to store the value of mshlflags in the marshalled data so as to be able to do the right thing at unmarshal time.

Argument	Type	Description
pstm	IStream *	the stream onto which the object should be marshalled. The stream passed to this function must be dynamically growable. It the absence of better information, it is suggested that this stream contain at least MARSHALINTERFACE_MIN bytes of space, though it is by no means guaranteed that this will be sufficient.
riid	REFIID	the interface that we wish to marshal.
pUnk	IUnknown *	the object on which we wish to marshal the interface riid.
dwDestContext	DWORD	the destination context in which the unmarshalling will occur.
pvDestContext	void*	related to some TBD destination contexts.
mshlflags	DWORD	the reason that the marshalling is taking place.
return value	HRESULT	S_OK, STG_E_MEDIUMFULL, E_FAIL

4.7.5.2. CoUnmarshalInterface

HRESULT CoUnmarshalInterface(pstm, iid, ppv)

Unmarshal from the given stream an object previously marshalled with CoMarshalInterface().

Argument	Type	Description
pstm	IStream *	the stream from which the object should be unmarshalled.
iid	REFIID	the interface with which we wish to talk to the reconstituted object.
ppv	void **	the place in which we should return the interface pointer.
return value	HRESULT	S_OK, E_FAIL

4.7.5.3. CoDisconnectObject

HRESULT CoDisconnectObject(pUnkInterface, dwReserved)

This function severs any extant Remote Procedure Call connections that are being maintained on behalf of all the interface pointers on this object. This is a very rude operation, and is not to be used in the normal course of processing; clients of interfaces should use IUnknown::Release() instead. In effect, this function is a privileged operation, which should generally only be invoked by the process in which the object actually is managed by the object implementation itself.

The primary purpose of this operation is to give an application process certain and definite control over remoting connections to other processes that may have been made from objects managed by the process. If the application process wishes to exit, then we do not want it to be the case that the extant reference counts from clients of the application's objects in fact keeps the process alive. When the application process wishes to exit, it should inform the extant clients of its objects[48] that the objects are going away. Having so informed its clients, the process can then call this function for each of the objects that it manages, even without waiting for a confirmation from each client. Having thus released resources maintained by the remoting connections, the application process can exit safely and cleanly. In effect, CoDisconnectObject() causes a controlled crash of the remoting connections to the object.

For illustration, contrast this with the situation with DDE. If it has extant DDE connections, an application is required to send a DDE Terminate message before exiting, and it is *also* responsible for waiting around for an acknowledgment from each client before it can actually exit. Thus, if the client process has crashed, the application process will wait around forever. Because of this, with DDE there simply is no way for an application process to reliably and robustly terminate itself. Using CoDisconnectObject(), we avoid this sort of situation.

Argument	Type	Description
punkInterface	IUnknown *	the object that we wish to disconnect. May be any interface on the object which is polymorphic with IUnknown*, not necessarily the exact interface returned by QueryInterface(IID_IUnknown...).
dwReserved	DWORD	reserved for future use; must be zero.
return value	HRESULT	S_OK, E_FAIL

4.7.5.4. CoReleaseMarshalData

HRESULT CoReleaseMarshalData(pstm)

This helper function destroys a previously marshalled data packet. This function must always be called in order to destroy data packets. Examples of when this occurs include:

1. an internal error during an RPC invocation prevented the UnmarshalInterface() operation from being attempted.
2. a marshalled-data-packet was removed from a global table.
3. following a successful, normal, unmarshal call.

This function works as should be expected: the class id is obtained from the stream; an instance is created; IMarshal is obtained from that instance; then IMarshal::ReleaseMarshalData() is invoked.

Argument	Type	Description
pstm	IStream*	a pointer to a stream that contains the data packet which is to be destroyed.
return value	HRESULT	S_OK, E_FAIL

4.7.5.5. CoGetStandardMarshal

HRESULT CoGetStandardMarshal(iid, pUnkObject, dwDestContext, pvDestContext, mshlflags, ppmarshal)

Return an IMarshal instance that knows how to do the Standard Marshalling and unmarshalling in order to create a proxy in the indicated destination context. Custom marshalling implementations should delegate to the marshaller here returned for destination contexts that they do not fully understand or which for which they choose not to take special action. The standard marshaller is also used in the case that the object being marshalled does not support Custom Marshalling.

This function is not implemented in OLE 2.0, though a non-functional "stub" of the function is indeed present. Programmers of custom marshallers should write their code as indicated so as to go ahead and call this stub anyway for destination contexts they

[48] using a higher-level notification scheme appropriate for the semantics of the particular connection. An example of this is OLE 2.0 is broadcasting IAdviseSink::OnClose() to connected links. See IOleObject::Close() for more details.

don't understand. With OLE 2.0, this is guaranteed to never happen (we only have one kind of destination context), so this will not cause problems. But by writing their code in this way, programmers of custom marshallers pre-enable themselves for the day when this is no longer the case (such as when networking support is added).

Argument	Type	Description
iid	REFIID	the interface id we would like to marshal.
pUnkObject	IUnknown*	the object that we wish to marshal. It is specifically *not* the case that this interface is known to be of shape iid; rather, it can be any interface on the object which conforms to IUnknown. The standard marshaller will internally do a QueryInterface().
dwDestContext	DWORD	the destination context in which the unmarshalling will occur.
pvDextContext	void*	associated with the destination context
mshlflags	DWORD	the marshal flags for the marshalling operation.
ppmarshal	IMarshal **	the place at which the standard marshaller should be returned.
return value	HRESULT	S_OK, E_FAIL

4.7.5.6. CoMarshalHresult

SCODE CoMarshalHresult(pstm, hresult)

Marshal an HRESULT to the given stream in such a way as it can be unmarshalled with CoUnmarshal-Hresult(). Custom marshallers should use this function when they have need to marshal an HRESULT.

Argument	Type	Description
pstm	IStream*	the stream into which the HRESULT is to be marshalled.
hresult	HRESULT	the HRESULT to be marshalled.
return value	HRESULT	S_OK; errors as in IStream::Write().

4.7.5.7. CoUnmarshalHresult

SCODE CoUnmarshalHresult(pstm, phresult)

Unmarshal an HRESULT previously marshalled with CoMarshalHresult(). Custom unmarshallers will want to use this function if the corresponding custom marshaller uses CoMarshalHresult().

Argument	Type	Description
pstm	IStream*	the stream into which the HRESULT is to be marshalled.
phresult	HRESULT*	the place at which the unmarshalled HRESULT is to be returned.
return value	HRESULT	S_OK; errors as in IStream::Read().

4.7.5.8. CoLockObjectExternal

HRESULT CoLockObjectExternal(pUnk, fLock, fLastUnlockReleases)

This function locks an object so that its reference count cannot decrement to zero. It also releases such a lock. From the object's point of view, the lock functionality is implemented by having the system AddRef() the object and not Release() it until CoLockObjectExternal(..., FALSE, ...) is later called.

CoLockObjectExternal() must be called in the process in which the object is actually resides (that is, the server process, not the process in which handlers for the object may be loaded).

The function can be used for the user's reference count as it acts external to the object, much like the user does. It can also be used for the IOleContainer::LockContainer() functionality, although the container must still keep a lock count so that it exits when the lock count reaches zero and the container is invisible.

This function does not in any way change the normal registration / revoking process for objects.

Argument	Type	Description
pUnk	IUnknown*	Points to the object to be locked or unlocked.

fLock	BOOL	Either locks or unlocks the object. FALSE releases such locks. TRUE holds the object alive (holds a reference to the object) independent of external or internal AddRef/Release operations, or registrations, or revokes. If fLock is TRUE, fLastLockReleases is ignored.
fLastLockReleases	BOOL	TRUE means release all pointers to the object if this lock is the last reference to the object which is supposed to hold it alive (there may be other references which are not supposed to hold it alive).

4.7.5.9. CoRegisterMessageFilter

HRESULT CoRegisterMessageFilter(lpMessageFilter, lplpMessageFilter);

This function is documented in the chapter on concurrency control.

4.7.6. IMarshal interface

IMarshal interface is the mechanism by which an object is custom-marshalled. IMarshal is defined as follows:

```
interface IMarshal : IUnknown {
    virtual    HRESULT    GetUnmarshalClass(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pclsid) = 0;
    virtual    HRESULT    GetMarshalSizeMax(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pcb) = 0;
    virtual    HRESULT    MarshalInterface(pstm, iid, pvInterface, dwDestContext, pvDestContext, mshlflags) = 0;
    virtual    HRESULT    UnmarshalInterface(pstm, iid, ppvInterface) = 0;
    virtual    HRESULT    DisconnectObject(dwReserved) = 0;
    virtual    HRESULT    ReleaseMarshalData(pstm) = 0;
    };
```

The process of Custom Marshalling an interface pointer involves two steps, with an optional third:

1. The code doing the marshalling calls IMarshal::GetUnmarshalClass(). This returns the class id that will be used to create an uninitialized proxy object in the unmarshalling context.
2. (optional) The marshaller calls IMarshal::GetMarshalSizeMax() to learn an upper bound on the amount of memory that will be required by the object to do the marshalling.
3. The marshaller calls IMarshal::MarshalInterface() to carry out the marshalling.

The class id and the bits that were marshalled into the stream are then conveyed by appropriate means to the destination, where they are unmarshalled. Unmarshalling involves the following essential steps:

1. Load the class object that corresponds to the class that the server said to use in GetUnmarshalClass().

```
IClassFactory * pcf;
CoGetClassObject(clsid, CLSCTX_HANDLER, IID_IClassFactory, &pcf);
```

2. Instantiate the class, asking for IMarshal interface:

```
IMarshal * proxy;
pcf->CreateInstance(NULL, IID_IMarshal, &proxy);
```

3. Initialize the proxy with IMarshal::UnmarshalInterface() using a copy of the bits that were originally produced by IMarshal::MarshalInterface() and asking for the interface that was originally marshalled.

```
IOriginal * pobj;
proxy->UnmarshalInterface(pstm, IID_Original, &pboj);
proxy->Release();
pcf->Release();
```

The object proxy is now ready for use.

4.7.6.1. IMarshal::GetUnmarshalClass

HRESULT IMarshal::GetUnmarshalClass(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pclsid)

Answer the class that should be used in the unmarshalling process to create an uninitialized object proxy.

dwDestContext is described in the API function CoMarshalInterface(). The implementation of GetUnmarshalClass() may wish for some destination contexts for which it takes no special action to delegate to the Standard Marshalling implementation, which is available through CoGetStandardMarshal(). In addition, this delegation should *always* be done if the dwDestContext parameter contains any flags that the GetUnmarshalClass() does not fully understand; it is by this means that we can extend the richness of destination contexts in the future. For example, in the future, one of these bits will likely be defined to indicate that the destination of the marshalling is across the network.

If the caller already has in hand the iid interface identified as being marshalled, he should pass the interface pointer through pvInterface. If he does not have this interface already, then he should pass NULL. This pointer will sometimes, though rarely, be used in order to determine the appropriate unmarshal class. If the IMarshal implementation really needs it, in can always QueryInterface() on itself to retrieve the interface pointer; we optionally pass it here only to improve efficiency.

Argument	Type	Description
iid	REFIID	the interface on this object that we are going to marshal.
pvInterface	void *	the actual pointer that will be marshalled. May be NULL.
dwDestContext	DWORD	the destination context relative to the current context in which the unmarshalling will be done.
pvDestContext	void*	non-NULL for some dwDestContext values.
mshlflags	DWORD	as in CoMarshalInterface().
pclsid	CLSID *	the class to be used in the unmarshalling process.
return value	HRESULT	S_OK, E_FAIL

4.7.6.2. IMarshal::MarshalInterface

HRESULT IMarshal::MarshalInterface(pstm, iid, pvInterface, dwDestContext, pvDestContext, mshlflags)

Marshal a reference to the interface iid of this object into the given stream. The interface actually marshalled is the one that would be returned by this->QueryInterface(iid, ...). Once the contents of this stream are conveyed to the destination by whatever means, the interface reference can be reconstituted by instantiating with IMarshal interface the class here retrievable with GetUnmarshalClass() and then calling IMarshal::UnmarshalInterface(). The implementation of IMarshal::MarshalInterface() writes in the stream any data required for initialization of this proxy.

If the caller already has in hand the iid interface identified as being marshalled, he should pass the interface pointer through pvInterface. If he does not have this interface already, then he should pass NULL; the IMarshal implementation will QueryInterface() on itself to retrieve the interface pointer.

On exit from this function, the seek pointer in the stream must be positioned immediately after the last byte of data written to the stream.

Argument	Type	Description
pstm	IStream *	the stream onto which the object should be marshaled.
iid	REFIID	the interface of this object that we wish to marshal.
pvInterface	void *	the actual pointer that will be marshalled. May be NULL.
dwDestContext	DWORD	as in CoMarshalInterface().
pvDestContext	void *	as in CoMarshalInterface().
mshlflags	DWORD	as in CoMarshalInterface().
return value	HRESULT	S_OK, STG_E_MEDIUMFULL, E_FAIL

4.7.6.3. IMarshal::GetMarshalSizeMax

HRESULT IMarshal::GetMarshalSizeMax(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pcb)

Return an upper bound on the amount of data that would be written into the marshalling stream in an IMarshal::MarshalInterface() stream. Callers can optionally use this value to pre-allocate stream buffers used in the marshalling process. Note that when IMarshal::MarshalInterface() is ultimately called, the IMarshal cannot rely on the caller actually having called GetMarshalSizeMax() beforehand; it must still be wary of STG_E_MEDIUMFULL errors returned by the stream.

The value returned by this function is only guaranteed to be valid so long as the internal state of the object being marshalled does not change. As a consequence, the actual marshalling should be done immediately after this function returns, or the caller runs the risk that the object requires more memory to marshal that it originally indicated.

An object *must* return a reasonable maximum size needed for marshalling: callers have the option of allocating a fixed-size marshalling buffer.

Argument	Type	Description
iid	REFIID	the interface of this object that we wish to marshal.
pvInterface	void *	the actual pointer that will be marshalled. May be NULL.
dwDestContext	DWORD	as in CoMarshalInterface().
pvDestContext	void *	as in CoMarshalInterface().
mshlflags	DWORD	as in CoMarshalInterface().
pcb	ULONG *	the place at which the maximum marshal size should be returned.
return value	HRESULT	S_OK

4.7.6.4. IMarshal::UnmarshalInterface

HRESULT IMarshal::UnmarshalInterface(pstm, iid, ppvInterface)

This is called as part of the unmarshalling process in order to initialize a newly created proxy; see the above sketch of the unmarshalling process for more details.

iid indicates the interface that the caller in fact would like to retrieve from this object; this interface instance is returned through ppvInterface. In order to support this, UnmarshalInterface() will often merely do a QueryInterface(iid, ppvInterface) on itself immediately before returning, though it is free to create a different object (an object with a different identity) if it wishes.

On successful exit from this function, the seek pointer must be positioned immediately after the data read from the stream. On error exit, the seek pointer should still be in this location: even in the face of an error, the stream should be positioned as if the unmarshal were successful.

See also CoReleaseMarshalData().

Argument	Type	Description
pstm	IStream *	the stream from which the interface should be unmarshalled.
iid	REFIID	the interface that the caller ultimately wants from the object.
ppvInterface	void **	the place at which the interface the caller wants is to be returned.
return value	HRESULT	S_OK, E_FAIL

4.7.6.5. IMarshal::Disconnect

HRESULT IMarshal::DisconnectObject(dwReserved)

This function is called by the implementation of CoDisconnectObject() in the event that the object attempting to be disconnected in fact supports Custom Marshalling. This is completely analogous to how CoMarshalInterface() defers to IMarshal::MarshalInterface() in if the object supports IMarshal.

Argument	Type	Description
dwReserved	DWORD	as in CoDisconnectObject().
return value	HRESULT	as in CoDisconnectObject().

4.7.6.6. IMarshal::ReleaseMarshalData

HRESULT IMarshal::ReleaseMarshalData(pstm)

This function is called by CoReleaseMarshalData() in order to actually carry out the destruction of a marshalled-data-packet. See that function for more details.

Note that whereas the IMarshal methods

GetUmarshalClass
GetMarshalSizeMax
MarshalInterface
Disconnect

are always called on the IMarshal interface instance in the originating side (server side), the method

UnmarshalInterface

is called on the receiving (client) side. (This should be no surprise.) However, the function

ReleaseMarshalData

(when needed) will be called on the receiving (client) side if the appropriate IMarshal instance can be successfully created there; otherwise, it is invoked on the originating (server) side.

Argument	Type	Description
pstm	IStream*	as in CoReleaseMarshalData().
return value	HRESULT	as in CoReleaseMarshalData().

4.8. Miscellaneous functions

The following are some miscellaneous utility functions associated with the other functionality discussed in this chapter.

```
HRESULT     StringFromIID(iid, lplpsz);
HRESULT     StringFromCLSID(clsid, lplpsz);
HRESULT     IIDFromString(lpsz, piid);
HRESULT     CLSIDFromString(lpsz, pcid);
BOOL        CoIsOle1Class(rclsid);
HRESULT     ProgIDFromCLSID(clsid, lplpszProgID);
HRESULT     CLSIDFromProgID(lpszProgID, lpclsid);

BOOL    IsEqualIID(iid1, iid2);
BOOL    IsEqualCLSID(iid1, iid2);

typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
    } FILETIME;

BOOL CoDosDateTimeToFileTime(wDOSDate, wDOSTime, lpft);
BOOL CoFileTimeToDosDateTime(lpft, lpwDOSDate, lpwDOSTime);
```

4.8.0.1. StringFromCLSID

HRESULT StringFromCLSID(clsid, lplpsz)

Convert the class clsid into a string of printable characters in such a way that different class ids always convert to different strings. This function is used by OLE in order to look up CLSIDs as keys in the registration database. The returned string is to be freed in the standard way, which is to use the task allocator; see CoGetMalloc().

Argument	Type	Description
clsid	REFCLSID	the class id of which we want a string representation.
lplpsz	LPSTR *	the place in which to return the resulting string.
return value	HRESULT	S_OK, E_NOMEMORY

4.8.0.2. CLSIDFromString

HRESULT CLSIDFromString(lpsz, pcid)

Turn a string generated by StringFromCLSID() back into the original CLSID.

Argument	Type	Description
lpsz	LPSTR	the string representation of the CLSID.
pcid	CLSID *	the place at which to return the CLSID.
return value	HRESULT	S_OK, E_NOMEMORY

4.8.0.3. IsEqualCLSID

BOOL IsEqualCLSID(clsid1, clsid2)

This function answers whether the two class ids are the same. This function is efficient.

Argument	Type	Description
iid1	REFCLSID	the one iid which is to be compared.
iid2	REFCLSID	the other one which is to be compared.
return value	BOOL	true if the two iids are the same; false otherwise.

4.8.0.4. StringFromIID

HRESULT StringFromIID(iid, lplpsz)

Convert the interface iid into a string of printable characters in such a way that different interface ids always convert to different strings. The returned string is to be freed in the standard way, which is to use the task allocator; see CoGetMalloc().

Argument	Type	Description
iid	REFIID	the interface of which we want a string representation.
lplpsz	LPSTR *	the place in which to return the resulting string.
return value	HRESULT	S_OK, E_NOMEMORY

4.8.0.5. IIDFromString

HRESULT IIDFromString(lpsz, piid)

Turn a string generated by StringFromIID() back into the original IID.

Argument	Type	Description
lpsz	LPSTR	the string representation of the IID.
piid	IID *	the place at which to return the IID.
return value	HRESULT	S_OK, E_NOMEMORY

4.8.0.6. IsEqualIID

BOOL IsEqualIID(iid1, iid2)

This function answers whether the two iids are the same. This function is efficient.

Argument	Type	Description
iid1	REFIID	the one iid which is to be compared.
iid2	REFIID	the other one which is to be compared.
return value	BOOL	true if the two iids are the same; false otherwise.

4.8.0.7. ProgIDFromCLSID

HRESULT ProgIDFromCLSID(clsid, lplpszProgID)

Every OLE2 class that belongs in an Insert Object dialog must have associated with it a "programmatic identifier" or ProgID. A ProgID is a string which uniquely identifies a given class. In addition to being used to determine eligibility for the Insert Object dialog, the ProgID can be used as an identifier in a "macro" programming language to identify a class. Finally, the ProgID is also the "classname" used for an OLE2 class when placed in OLE1 containers.

The ProgID string must:

- have no more than 39 characters (i.e.: 39 is OK).
- contain no punctuation (including underscore), with the one exception that it may contain a single period.
- not start with a digit
- be different from the class name of any OLE1 application, *including the OLE1 version of the same application*, if there is one.

CLSIDFromProgID() and ProgIdFromCLSID() can convert back and forth between the two representations. These functions use the registration database do the conversion; see the appendix for details. OLE2 application authors are responsible for correctly configuring the registration database at application installation time.

The ProgID must *never* be shown to the user in the user interface. If you need a short human-readable string for an object, call IOleObject::GetUserType(USERCLASSTYPE_SHORT, &szShortName).

Argument	Type	Description
clsid	REFCLSID	the class whose ProgID is requested.
lplpszProgID	LPSTR *	the place at which the ProgID is returned. As usual, the task allocator is used; see CoGetMalloc().
return value	HRESULT	S_OK, CO_E_CLASSNOTREG, CO_E_READREGDB

4.8.0.8. CLSIDFromProgID

HRESULT CLSIDFromProgID(lpszProgID, lpclsid)

Given a ProgID, return its associated CLSID. If the ProgID is in fact the class identifier of a real OLE1 class, then this function will automatically synthesize a CLSID for it.

Argument	Type	Description
lpszProgID	LPSTR	the ProgID whose CLSID is requested.
lpclsid	CLSID *	the place at which the CLSID is returned.
return value	HRESULT	S_OK, E_INVALIDARG, CO_E_CLASSSTRING (if the registered clsid for the ProgID is invalid), CO_E_WRITEREGDB

4.8.0.9. CoDosDateTimeToFileTime

BOOL CoDosDateTimeToFileTime(wDOSDate, wDOSTime, lpft)

This function converts between the DOS representation of time and date and a representation known as a FILETIME, which is a 64-bit value representing the number of 100-nanosecond intervals since January 1st, 1601. (This date was chosen because it is the start of a new quadricentury.) The FILETIME structure was

originally defined as part of the Win32 definition. On non-Win32 platforms, it is provided as part of OLE. The CoDosDateTimeToFileTime() function has the same semantics as the Win32 function DosDateTime-ToFileTime().

MS-DOS records file dates and times as packed 16-bit values. An MS-DOS *date* has the following format:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, etc.)
9-15	Year offset from 1980 (add 1980 to get actual year)

An MS-DOS *time* has the following format:

Bits	Contents
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

The CoDosDateTimeToFileTime() function converts MS-DOS date and time values into FILETIME values. The CoFileTimeToDosDateTime() function does the reverse, converting FILETIME values into MS-DOS date and time values.

Argument	Type	Description
wDOSDate	WORD	16-bit MS-DOS date
wDOSTime	WORD	16-bit MS-DOS time
lpft	LPFILETIME	address of buffer for 64-bit file time
return value	BOOL	success or failure.

4.8.0.10. CoFileTimeToDosDateTime

BOOL CoFileTimeToDosDateTime(lpft, lpwDOSDate, lpwDOSTime)

The CoFileTimeToDosDateTime() function converts a 64-bit file time into MS-DOS date and time values. This is the inverse operation to that provided by the CoDosDateTimeToFileTime() function.

The CoFileTimeToDosDateTime() function has the same semantics as the Win32 function FileTimeTo-DosDateTime().

Argument	Type	Description
lpft	FILETIME*	address of buffer for 64-bit file time
lpwDOSDate	WORD*	16-bit MS-DOS date
lpwDOSTime	WORD*	16-bit MS-DOS time
return value	BOOL	success or failure. Failure will be due to invalid arguments.

4.8.0.11. CoFileTimeNow

HRESULT CoFileTimeNow(lpFileTime)

Returns the current time as a FILETIME.

Argument	Type	Description
lpFileTime	FILETIME*	the place to return the current time
return value	HRESULT	S_OK