

This chapter describes the data type and interface mapping between COM and CORBA. The mappings are described in the context of both Win16 and Win32 COM due to the differences between the versions of COM and between the automated tools available to COM developers under these environments. The mapping is designed to be fully implemented by automated interworking tools.

Contents

This chapter contains the following sections.

Section Title	Page
“Data Type Mapping”	18-1
“CORBA to COM Data Type Mapping”	18-2
“COM to CORBA Data Type Mapping”	18-33

18.1 Data Type Mapping

The data type model used in this mapping for Win32 COM is derived from MIDL (a derivative of DCE IDL). COM interfaces using “custom marshaling” must be hand-coded and require special treatment to interoperate with CORBA using automated tools. This specification does not address interworking between CORBA and custom-marshaled COM interfaces.

The data type model used in this mapping for Win16 COM is derived from ODL since Microsoft RPC and the Microsoft MIDL compiler are not available for Win16. The ODL data type model was chosen since it is the only standard, high-level representation available to COM object developers on Win16.

Note that although the MIDL and ODL data type models are used as the reference for the data model mapping, there is no requirement that either MIDL or ODL be used to implement a COM/CORBA interworking solution.

In many cases, there is a one-to-one mapping between COM and CORBA data types. However, in cases without exact mappings, run-time conversion errors may occur. Conversion errors will be discussed in Mapping for Exception Types under Section 18.2.10, “Interface Mapping,” on page 18-11.

18.2 CORBA to COM Data Type Mapping

18.2.1 Mapping for Basic Data Types

The basic data types available in OMG IDL map to the corresponding data types available in Microsoft IDL as shown in Table 18-1.

Table 18-1 OMG IDL to MIDL Intrinsic Data Type Mappings

OMG IDL	Microsoft IDL	Microsoft ODL	Description
short	short	short	Signed integer with a range of $-2^{15} \dots 2^{15} - 1$
long	long	long	Signed integer with a range of $-2^{31} \dots 2^{31} - 1$
unsigned short	unsigned short	unsigned short	Unsigned integer with a range of $0 \dots 2^{16} - 1$
unsigned long	unsigned long	unsigned long	Unsigned integer with a range of $0 \dots 2^{32} - 1$
float	float	float	IEEE single-precision floating point number
double	double	double	IEEE double-precision floating point number
char	char	char	8-bit quantity limited to the ISO Latin-1 character set
wchar	WCHAR	WCHAR	wide character
boolean	boolean	boolean	8-bit quantity that is limited to 1 and 0
octet	byte	unsigned char	8-bit opaque data type, guaranteed to not undergo any conversion during transfer between systems.

Note – midl and mktyplib disagree about the size of boolean when used in an ODL specification. To avoid this ambiguity, we make the mapping explicit and use the VARIANT BOOL type instead of the built-in boolean type.

18.2.2 Mapping for Constants

The mapping of the OMG IDL keyword **const** to Microsoft IDL and ODL is almost exactly the same. The following are the OMG IDL definitions for constants:

```

// OMG IDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const float F = ...;
const char C = ...;
const boolean B = ...;
const string STR = "...";

```

that map to the following Microsoft IDL and ODL definitions for constants:

```

// Microsoft IDL and ODL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const float F = ...;
const char C = ...;
const boolean B = ...;
const string STR = "...";

```

18.2.3 Mapping for Enumerators

CORBA has enumerators that are not explicitly tagged with values. Microsoft IDL and ODL support enumerators that are explicitly tagged with values. The constraint is that any language mapping that permits two enumerators to be compared or defines successor or predecessor functions on enumerators must conform to the ordering of the enumerators as specified in the OMG IDL.

```

// OMG IDL
interface MyInft {
    enum A_or_B_or_C {A, B, C};
};

```

CORBA enumerators are mapped to COM enumerations directly according to CORBA C language binding. The Microsoft IDL keyword `v1_enum` is required in order for an enumeration to be transmitted as 32-bit values. Microsoft recommends that this keyword be used on 32-bit platforms, since it increases the efficiency of marshalling and unmarshalling data when such an enumerator is embedded in a structure or union.

```

// Microsoft IDL and ODL
uuid(...),
interface IMyIntf {
    typedef [v1_enum]
    enum tagA or B or C {MyIntf A = 0,
                        MyInft B,
                        MyIntf C }
};

```

```

                                MyIntf A or B or C;
};

```

A maximum of 2^{32} identifiers may be specified in an enumeration in CORBA. Enumerators in Microsoft IDL and ODL will only support 2^{16} identifiers, and therefore, truncation may result.

18.2.4 Mapping for String Types

CORBA currently defines the data type **string** to represent strings that consist of 8-bit quantities, which are NULL-terminated.

Microsoft IDL and ODL define a number of different data types, which are used to represent both 8-bit character strings and strings containing wide characters based on Unicode.

Table 18-2 illustrates how to map the string data types in OMG IDL to their corresponding data types in both Microsoft IDL and ODL.

Table 18-2 OMG IDL to Microsoft IDL/ODL String Mappings

OMG IDL	Microsoft IDL	Microsoft ODL	Description
string	LPSTR [string,unique] char *	LPSTR	Null-terminated 8-bit character string
wstring	LPWSTR [string,unique] wchar t *	LPWSTR	Null-terminated Unicode string

OMG IDL supports two different types of strings: *bounded* and *unbounded*. Bounded strings are defined as strings that have a maximum length specified; whereas, unbounded strings do not have a maximum length specified.

18.2.4.1 Mapping for Unbounded String Types

The definition of an unbounded string limited to 8-bit quantities in OMG IDL

```

// OMG IDL
typedef string UNBOUNDED_STRING;

```

is mapped to the following syntax in Microsoft IDL and ODL, which denotes the type of a “stringified unique pointer to character.”

```

// Microsoft IDL and ODL
typedef [string, unique] char * UNBOUNDED_STRING;

```

In other words, a value of type **UNBOUNDED_STRING** is a non-NULL pointer to a one-dimensional null-terminated character array whose extent and number of valid elements can vary at run-time.

18.2.4.2 Mapping for Bounded String Types

Bounded strings have a slightly different mapping between OMG IDL and Microsoft IDL and ODL. The following OMG IDL definition for a bounded string:

```
// OMG IDL
const long N = ...;
typedef string<N> BOUNDED_STRING;
```

maps to the following syntax in Microsoft IDL and ODL for a “stringified non-conformant array.”

```
// Microsoft IDL and ODL
const long N = ... ;
typedef [string, unique] char (* BOUNDED_STRING) [N];
```

In other words, the encoding for a value of type **BOUNDED_STRING** is that of a null-terminated array of characters whose extent is known at compile time, and the number of valid characters can vary at run-time.

18.2.5 Mapping for Struct Types

OMG IDL uses the keyword `struct` to define a record type, consisting of an ordered set of name-value pairs representing the member types and names. A structure defined in OMG IDL maps bidirectionally to Microsoft IDL and ODL structures. Each member of the structure is mapped according to the mapping rules for that data type.

An OMG IDL struct type with members of types T0, T1, T2, and so on

```
// OMG IDL
typedef ... T0
typedef ... T1;
typedef ... T2;
...
typedef ... Tn;
struct STRUCTURE
{
    T0 m0;
    T1 m1;
    T2 m2;
...
    Tn mN;
};
```

has an encoding equivalent to a Microsoft IDL and ODL structure definition, as follows.

```
// Microsoft IDL and ODL
typedef ... T0;
typedef ... T1;
typedef ... T2;
```

```

...
typedef ... Tn;
typedef struct
{
    T0 m0;
    T1 m1;
    T2 m2;
    ...
    TN mN;
} STRUCTURE;

```

Self-referential data types are expanded in the same manner. For example,

```

struct A { // OMG IDL
    sequence<A> v1;
};

```

is mapped as

```

typedef struct A {
    struct { // MIDL
        unsigned long cbMaxSize;
        unsigned long cbLengthUsed;
        [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
        struct A * pValue;
    } v1;
} A;

```

18.2.6 Mapping for Union Types

OMG IDL defines unions to be encapsulated discriminated unions: the discriminator itself must be encapsulated within the union.

In addition, the OMG IDL union discriminants must be constant expressions. The discriminator tag must be a previously defined **long**, **short**, **unsigned long**, **unsigned short**, **char**, **boolean**, or **enum** constant. The default case can appear at most once in the definition of a discriminated union, and case labels must match or be automatically castable to the defined type of the discriminator.

The following definition for a discriminated union in OMG IDL

```

// OMG IDL
enum UNION_DISCRIMINATOR
{
    dChar=0,
    dShort,
    dLong,
    dFloat,
    dDouble
};

union UNION_OF_CHAR_AND_ARITHMETIC
switch(UNION_DISCRIMINATOR)
{
    case dChar: char c;
    case dShort: short s;
    case dLong: long l;
    case dFloat: float f;
    case dDouble: double d;
    default: octet v[8];
};

```

is mapped into encapsulated unions in Microsoft IDL as follows:

```

// Microsoft IDL
typedef enum [v1 enum]
{
    dchar=0,
    dShort,
    dLong,
    dFloat,
    dDouble
} UNION_DISCRIMINATOR;

typedef union switch (UNION_DISCRIMINATOR DCE_d)
{
    case dChar: char c;
    case dShort: short s;
    case dLong: long l;
    case dFloat: float f;
    case dDouble: double d;
    default: byte v[8];
}UNION_OF_CHAR_AND_ARITH

```

18.2.7 Mapping for Sequence Types

OMG IDL defines the keyword **sequence** to be a one-dimensional array with two characteristics: an optional maximum size that is fixed at compile time, and a length that is determined at run-time. Like the definition of strings, OMG IDL allows sequences to be defined in one of two ways: bounded and unbounded. A sequence is bounded if a maximum size is specified, else it is considered unbounded.

18.2.7.1 Mapping for Unbounded Sequence Types

The mapping of the following OMG IDL syntax for the unbounded sequence of type **T**

```
// OMG IDL for T
typedef ... T;
typedef sequence<T> UNBOUNDED_SEQUENCE;
```

maps to the following Microsoft IDL and ODL syntax:

```
// Microsoft IDL or ODL
typedef ... U;
typedef struct
{
    unsigned long cbMaxSize;
    unsigned long cbLengthUsed;
    [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
        U * pValue;
    } UNBOUNDED_SEQUENCE;
```

The encoding for an unbounded OMG IDL sequence of type **T** is that of a Microsoft IDL or ODL struct containing a unique pointer to a conformant array of type **U**, where **U** is the Microsoft IDL or ODL mapping of **T**. The enclosing struct in the Microsoft IDL/ODL mapping is necessary to provide a scope in which extent and data bounds can be defined.

18.2.7.2 Mapping for Bounded Sequence Types

The mapping for the following OMG IDL syntax for the bounded sequence of type **T** that can grow to be **N** size:

```
// OMG IDL for T
const long N = ...;
typedef ...T;
typedef sequence<T,N> BOUNDED_SEQUENCE_OF_N;
```

maps to the following Microsoft IDL or ODL syntax:

```
// Microsoft IDL or ODL
const long N = ...;
typedef ...U;
```

```
typedef struct
{
    unsigned long reserved;
    unsigned long cbLengthUsed;
    [length_is(cbLengthUsed)] U Value[N];
} BOUNDED_SEQUENCE_OF_N;
```

Note – The maximum size of the bounded sequence is declared in the declaration of the array and therefore a [size is ()] attribute is not needed.

18.2.8 Mapping for Array Types

OMG IDL arrays are fixed length multidimensional arrays. Both Microsoft IDL and ODL also support fixed length multidimensional arrays. Arrays defined in OMG IDL map bidirectionally to COM fixed length arrays. The type of the array elements is mapped according to the data type mapping rules.

The mapping for an OMG IDL array of some type T is that of an array of the type U as defined in Microsoft IDL and ODL, where U is the result of mapping the OMG IDL T into Microsoft IDL or ODL.

```
// OMG IDL for T
const long N = ...;
typedef ... T;
typedef T ARRAY_OF_T[N];

// Microsoft IDL or ODL for T
const long N = ...;
typedef ... U;
typedef U ARRAY_OF_U[N];
```

In Microsoft IDL and ODL, the name **ARRAY_OF_U** denotes the type of a “one-dimensional nonconformant and nonvarying array of U.” The value N can be of any integral type, and const means (as in OMG IDL) that the value of N is fixed and known at IDL compilation time. The generalization to multidimensional arrays follows the obvious mapping of syntax.

Note that if the ellipsis were **octet** in the OMG IDL, then the ellipsis would have to be **byte** in Microsoft IDL or ODL. That is why the types of the array elements have different names in the two texts.

18.2.9 Mapping for the *any* Type

The CORBA **any** type permits the specification of values that can express any OMG IDL data type. There is no direct or simple mapping of this type into COM, thus we map it to the following interface definition:

```
// Microsoft IDL
typedef [v1_enum] enum CORBAAnyDataTagEnum {
```

```

        anySimpleValTag,
        anyAnyValTag,
        anySeqValTag,
        anyStructValTag,
        anyUnionValTag
    } CORBAAnyDataTag;

typedef union CORBAAnyDataUnion switch(CORBAAnyDataTag
    whichOne){
    case anyAnyValTag:
        ICORBA_Any *anyVal;
    case anySeqValTag:
    case anyStructValTag:
        struct {
            [string, unique] char * repositoryId;
            unsigned long cbMaxSize;
            unsigned long cbLengthUsed;
            [size_is(cbMaxSize), length_is(cbLengthUsed),
            unique]
            union CORBAAnyDataUnion *pVal;
        } multiVal;
    case anyUnionValTag:
        struct {
            [string, unique] char * repositoryId;
            long disc;
            union CORBAAnyDataUnion *value;
        } unionVal;
    case anyObjectValTag:
        struct {
            [string, unique] char * repositoryId;
            VARIANT val;
        } objectVal;
    case anySimpleValTag: // All other types
        VARIANT simpleVal;
    } CORBAAnyData;

.... uuid(74105F50-3C68-11cf-9588-AA0004004A09) ]
interface ICORBA_Any: IUnknown
{
    HRESULT _get_value([out] VARIANT * val );
    HRESULT _put_value([in] VARIANT val );
    HRESULT _get_CORBAAnyData([out] CORBAAnyData* val);
    HRESULT _put_CORBAAnyData([in] CORBAAnyData val );
    HRESULT _get_typeCode([out] ICORBA_TypeCode ** tc);
}

```

In most cases, a COM client can use the `_get_value()` or `_put_value()` method to set and get the value of a CORBA **any**. However, the data types supported by a VARIANT are too restrictive to support all values representable in an **any**, such as structs and unions. In cases where the data types can be represented in a VARIANT, they will be; in other cases, they will optionally be returned as an **IStream** pointer

in the VARIANT. An implementation may choose not to represent these types as an **IStream**, in which case an SCODE value of E_DATA_CONVERSION is returned when the VARIANT is requested.

18.2.10 Interface Mapping

18.2.10.1 Mapping for interface identifiers

Interface identifiers are used in both CORBA and COM to uniquely identify interfaces. These allow the client code to retrieve information about, or to inquire about, other interfaces of an object.

CORBA identifies interfaces using the RepositoryId. The RepositoryId is a unique identifier for, among other things, an interface. COM identifies interfaces using a structure similar to the DCE UUID (in fact, identical to a DCE UUID on Win32) known as an IID. As with CORBA, COM specifies that the textual names of interfaces are only for convenience and need not be globally unique.

The CORBA RepositoryId is mapped, bidirectionally, to the COM IID. The algorithm for creating the mapping is detailed in Section 17.5.4, “Mapping Interface Identity,” on page 17-16.

18.2.10.2 Mapping for exception types

The CORBA object model uses the concept of exceptions to report error information. Additional, exception-specification information may accompany the exception. The exception-specific information is a specialized form of a record. Because it is defined as a record, the additional information may consist of any of the basic data types or a complex data type constructed from one or more basic data types. Exceptions are classified into two types: System (Standard) Exceptions and User Exceptions.

COM provides error information to clients only if an operation uses a return result of type HRESULT. A COM HRESULT with a value of zero indicates success. The HRESULT then can be converted into an SCODE (the SCODE is explicitly specified as being the same as the HRESULT on Win32 platforms). The SCODE can then be examined to determine whether the call succeeded or failed. The error or success code, also contained within the SCODE, is composed of a “facility” major code (13 bits on Win32 and 4 bits on Win16) and a 16-bit minor code.

Unlike CORBA, COM provides no standard way to return user-defined exception data to the client. Also, there is no standard mechanism in COM to specify the completion status of an invocation. In addition, it is not possible to predetermine what set of errors a COM interface might return based on the definition of the interface as specified in Microsoft IDL, ODL, or in a type library. Although the set of status codes that can be returned from a COM operation must be fixed when the operation is defined, there is currently no machine-readable way to discover the set of valid codes.

Since the CORBA exception model is significantly richer than the COM exception model, mapping CORBA exceptions to COM requires an additional protocol to be defined for COM. However, this protocol does not violate backwards compatibility, nor does it require any changes to COM. To return the User Exception data to a COM client, an optional parameter is added to the end of a COM operation signature when mapping CORBA operations, which raise User Exceptions. System exception information is returned in a standard OLE Error Object.

Mapping for System Exceptions

System exceptions are standard exception types, which are defined by the CORBA specification and are used by the Object Request Broker (ORB) and object adapters (OA). Standard exceptions may be returned as a result of any operation invocation, regardless of the interface on which the requested operation was attempted.

There are two aspects to the mapping of System Exceptions. One aspect is generating an appropriate HRESULT for the operation to return. The other aspect is conveying System Exception information via a standard OLE Error Object.

The following table shows the HRESULT, which must be returned by the COM View when a CORBA System Exception is raised. Each of the CORBA System Exceptions is assigned a 16-bit numerical ID starting at 0x200 to be used as the code (lower 16 bits) of the HRESULT. Because these errors are interface-specific, the COM facility code FACILITY_ITF is used as the facility code in the HRESULT.

Bits 12-13 of the HRESULT contain a bit mask, which indicates the completion status of the CORBA request. The bit value 00 indicates that the operation did not complete, a bit value of 01 indicates that the operation did complete, and a bit value of 02 indicates that the operation may have completed. Table 18-3 lists the HRESULT constants and their values.

Table 18-3 Standard Exception to SCODE Mapping

HRESULT Constant	HRESULT Value
ITF_E_UNKNOWN_NO	0x40200
ITF_E_UNKNOWN_YES	0x41200
ITF_E_UNKNOWN_MAYBE	0x42200
ITF_E_BAD_PARAM_NO	0x40201
ITF_E_BAD_PARAM_YES	0x41201
ITF_E_BAD_PARAM_MAYBE	0x42201
ITF_E_NO_MEMORY_NO	0x40202
ITF_E_NO_MEMORY_YES	0x41202
ITF_E_NO_MEMORY_MAYBE	0x42202
ITF_E_IMP_LIMIT_NO	0x40203

Table 18-3 Standard Exception to SCODE Mapping (Continued)

ITF_E_IMP_LIMIT_YES	0x41203
ITF_E_IMP_LIMIT_MAYBE	0x42203
ITF_E_COMM_FAILURE_NO	0x40204
ITF_E_COMM_FAILURE_YES	0x41204
ITF_E_COMM_FAILURE_MAYBE	0x42204
ITF_E_INV_OBJREF_NO	0x40205
ITF_E_INV_OBJREF_YES	0x41205
ITF_E_INV_OBJREF_MAYBE	0x42205
ITF_E_NO_PERMISSION_NO	0x40206
ITF_E_NO_PERMISSION_YES	0x41206
ITF_E_NO_PERMISSION_MAYBE	0x42206
ITF_E_INTERNAL_NO	0x40207
ITF_E_INTERNAL_YES	0x41207
ITF_E_INTERNAL_MAYBE	0x42207
ITF_E_MARSHAL_NO	0x40208
ITF_E_MARSHAL_YES	0x41208
ITF_E_MARSHAL_MAYBE	0x42208
ITF_E_INITIALIZE_NO	0x40209
ITF_E_INITIALIZE_YES	0x41209
ITF_E_INITIALIZE_MAYBE	0x42209
ITF_E_NO_IMPLEMENT_NO	0x4020A
ITF_E_NO_IMPLEMENT_YES	0x4120A
ITF_E_NO_IMPLEMENT_MAYBE	0x4220A
ITF_E_BAD_TYPECODE_NO	0x4020B
ITF_E_BAD_TYPECODE_YES	0x4120B
ITF_E_BAD_TYPECODE_MAYBE	0x4220B
ITF_E_BAD_OPERATION_NO	0x4020C
ITF_E_BAD_OPERATION_YES	0x4120C
ITF_E_BAD_OPERATION_MAYBE	0x4220C

Table 18-3 Standard Exception to SCODE Mapping (Continued)

ITF_E_NO_RESOURCES_NO	0x4020D
ITF_E_NO_RESOURCES_YES	0x4120D
ITF_E_NO_RESOURCES_MAYBE	0x4220D
ITF_E_NO_RESPONSE_NO	0x4020E
ITF_E_NO_RESPONSE_YES	0x4120E
ITF_E_NO_RESPONSE_MAYBE	0x4220E
ITF_E_PERSIST_STORE_NO	0x4020F
ITF_E_PERSIST_STORE_YES	0x4120F
ITF_E_PERSIST_STORE_MAYBE	0x4220F
ITF_E_BAD_INV_ORDER_NO	0x40210
ITF_E_BAD_INV_ORDER_YES	0x41210
ITF_E_BAD_INV_ORDER_MAYBE	0x42210
ITF_E_TRANSIENT_NO	0x40211
ITF_E_TRANSIENT_YES	0x41211
ITF_E_TRANSIENT_MAYBE	0x42211
ITF_E_FREE_MEM_NO	0x40212
ITF_E_FREE_MEM_YES	0x41212
ITF_E_FREE_MEM_MAYBE	0x42212
ITF_E_INV_IDENT_NO	0x40213
ITF_E_INV_IDENT_YES	0x41213
ITF_E_INV_IDENT_MAYBE	0x42213
ITF_E_INV_FLAG_NO	0x40214
ITF_E_INV_FLAG_YES	0x41214
ITF_E_INV_FLAG_MAYBE	0x42214
ITF_E_INTF_REPOS_NO	0x40215
ITF_E_INTF_REPOS_YES	0x41215
ITF_E_INTF_REPOS_MAYBE	0x42215
ITF_E_BAD_CONTEXT_NO	0x40216
ITF_E_BAD_CONTEXT_YES	0x41216

Table 18-3 Standard Exception to SCODE Mapping (Continued)

ITF_E_BAD_CONTEXT_MAYBE	0x42216
ITF_E_OBJ_ADAPTER_NO	0x40217
ITF_E_OBJ_ADAPTER_YES	0x41217
ITF_E_OBJ_ADAPTER_MAYBE	0x42217
ITF_E_DATA_CONVERSION_NO	0x40218
ITF_E_DATA_CONVERSION_YES	0x41218
ITF_E_DATA_CONVERSION_MAYBE	0x42218
ITF_E_OBJ_NOT_EXIST_NO	0X40219
ITF_E_OBJ_NOT_EXIST_MAYBE	0X41219
ITF_E_OBJ_NOT_EXIST_YES	0X42219
ITF_E_TRANSACTION_REQUIRED_NO	0x40220
ITF_E_TRANSACTION_REQUIRED_MAYBE	0x41220
ITF_E_TRANSACTION_REQUIRED_YES	0x42220
ITF_E_TRANSACTION_ROLLEDBACK_NO	0x40221
ITF_E_TRANSACTION_ROLLEDBACK_MAYBE	0x41221
ITF_E_TRANSACTION_ROLLEDBACK_YES	0x42221
ITF_E_INVALID_TRANSACTION_NO	0x40222
ITF_E_INVALID_TRANSACTION_MAYBE	0x41222
ITF_E_INVALID_TRANSACTION_YES	0x42222

It is not possible to map a System Exception's minor code and RepositoryId into the HRESULT. Therefore, OLE Error Objects may be used to convey these data. Writing the exception information to an OLE Error Object is optional. However, if the Error Object is used for this purpose, it must be done according to the following specifications.

- The COM View must implement the standard COM interface ISupportErrorInfo such that the View can respond affirmatively to an inquiry from the client as to whether Error Objects are supported by the View Interface.
- The COM View must call SetErrorInfo with a NULL value for the IErrorInfo pointer parameter when the mapped CORBA operation is completed without an exception being raised. Calling **SetErrorInfo** in this fashion assures that the Error Object on that thread is thoroughly destroyed.

The properties of the OLE Error Object must be set according to Table 18-4.

Table 18-4 Error Object Usage for CORBA System Exceptions

Property	Description
bstrSource	<interface name>.<operation name> where the interface and operation names are those of the CORBA interface that this Automation View is representing.
bstrDescription	CORBA System Exception: [<exception repository id>] minor code [<minor code>][<completion status>] where the <exception repository id> and <minor code> are those of the CORBA system exception. <completion status> is “YES,” “NO,” or “MAYBE” based upon the value of the system exception’s CORBA completion status. Spaces and square brackets are literals and must be included in the string.
bstrHelpFile	Unspecified
dwHelpContext	Unspecified
GUID	The IID of the COM View Interface

A COM View supporting error objects would have code, which approximates the following C++ example.

```
SetErrorInfo(OL,NULL); // Initialize the thread-local error
object
try
{
    // Call the CORBA operation
}
catch(...)
{
    ...

    CreateErrorInfo(&pICreateErrorInfo);
    pICreateErrorInfo->SetSource(...);
    pICreateErrorInfo->SetDescription(...);
    pICreateErrorInfo->SetGUID(...);
    pICreateErrorInfo
->QueryInterface(IID_IErrorInfo,&pIErrorInfo);
    pICreateErrorInfo->SetErrorInfo(OL,pIErrorInfo);
    pIErrorInfo->Release();
    pICreateErrorInfo->Release();

    ...
}
```

A client to a COM View would access the OLE Error Object with code approximating the following.

```

// After obtaining a pointer to an interface on
// the COM View, the
// client does the following one time

pIMyMappedInterface->QueryInterface(IID_ISupportErrorInfo,
                                   &pISupportErrorInfo);

hr = pISupportErrorInfo
     ->InterfaceSupportsError-
Info(IID_MyMappedInterface);
BOOL bSupportsErrorInfo = (hr == NOERROR ? TRUE : FALSE);
...
// Call to the COM operation...
HRESULT hrOperation = pIMyMappedInterface->...

if (bSupportsErrorInfo)
{
    HRESULT hr = GetErrorInfo(0,&pIErrorInfo);

    // S_FALSE means that error data is not available,
    NO_ERROR
    // means it is
    if (hr == NO_ERROR)
    {
        pIErrorInfo->GetSource(...);

        // Has repository id & minor code. hrOperation
(above)
        // has the completion status encoded into it.
        pIErrorInfo->GetDescription(...);
    }
}

```

Mapping for User Exception Types

User exceptions are defined by users in OMG IDL and used by the methods in an object server to report operation-specific errors. The definition of a User Exception is identified in an OMG IDL file with the keyword `exception`. The body of a User Exception is described using the syntax for describing a structure in OMG IDL.

When CORBA User Exceptions are mapped into COM, a structure is used to describe various information about the exception — hereafter called an Exception structure. The structure contains members, which indicate the type of the CORBA exception, the identifier of the exception definition in a CORBA Interface Repository, and interface pointers to User Exceptions. If an interface raises a user exception, a structure is constructed whose name is the interface name [fully scoped] followed by “Exceptions.” For example, if an operation in **MyModule:MyInterface** raises a user exception, then there will be a structure created named **MyModule_MyInterfaceExceptions**.

A template illustrating this naming convention is as follows.

```

// Microsoft IDL and ODL
typedef enum { NO_EXCEPTION, USER_EXCEPTION}
             ExceptionType;

typedef struct
{
    ExceptionType    type;
    LPTSTR           repositoryId;
    I<ModuleName_InterfaceName>UserException
        *...piUserException;
} <ModuleName_InterfaceName>Exceptions;

```

The Exceptions structure is specified as an output parameter, which appears as the last parameter of any operation mapped from OMG IDL to Microsoft IDL, which raises a User Exception. The Exceptions structure is always passed by indirect reference. Because of the memory management rules of COM, passing the Exceptions structure as an output parameter by indirect reference allows the parameter to be treated as optional by the callee¹. The following example illustrates this point.

```

// Microsoft IDL
interface IBANKAccount
{
    HRESULT Withdraw(           [in] float fAmount,
                               [out] float pfNewBalance,
                               [out] BANK_AccountExceptions
                               ** pException);
};

```

The caller can indicate that no exception information should be returned, if an exception occurs, by specifying NULL as the value for the Exceptions parameter of the operation. If the caller expects to receive exception information, it must pass the address of a pointer to the memory in which the exception information is to be placed. COM's memory management rules state that it is the responsibility of the caller to release this memory when it is no longer required.

If the caller provides a non-NULL value for the Exceptions parameter and the callee is to return exception information, the callee is responsible for allocating any memory used to hold the exception information being returned. If no exception is to be returned, the callee need do nothing with the parameter value.

If a CORBA exception is not raised, then S_OK must be returned as the value of the HRESULT to the callee, indicating the operation succeeded. The value of the HRESULT returned to the callee when a CORBA exception has been raised depends upon the type of exception being raised and whether an Exception structure was specified by the caller.

1. Vendors that map the MIDL definition directly to C++ should map the exception struct parameter as defaulting to a NULL pointer.

The following OMG IDL statements show the definition of the format used to represent User Exceptions.

```
// OMG IDL
module BANK
{
  ...
  exception InsuffFunds { float balance };
  exception InvalidAmount { float amount };
  ...
  interface Account
  {
    exception NotAuthorized { };
    float Deposit( in float Amount )
      raises( InvalidAmount );
    float Withdraw( in float Amount )
      raises( InvalidAmount, NotAuthorized );
  };
};
```

and map to the following statements in Microsoft IDL and ODL.

```
// Microsoft IDL and ODL
struct Bank_InsuffFunds
{
  float balance;
};

struct Bank_InvalidAmount
{
  float amount;
};

struct BANK_Account_NotAuthorized
{
};

interface IBANK_AccountUserExceptions : IUnknown
{
  HRESULT _get_InsuffFunds( [out] BANK_InsuffFunds
    * exceptionBody );
  HRESULT _get_InvalidAmount( [out] BANK_InvalidAmount
    * exceptionBody );
  HRESULT _get_NotAuthorized( [out]
    BANK_Account_NotAuthorized
    * exceptionBody );
};

typedef struct
{
```

```

        ExceptionType      type;
        LPTSTR             repositoryId;
        IBANK_AccountUserExceptions * piUserException;
    } BANK_AccountExceptions;

```

User exceptions are mapped to a COM interface and a structure that describes the body of information to be returned for the User Exception. A COM interface is defined for each CORBA interface containing an operation that raises a User Exception. The name of the interface defined for accessing User Exception information is constructed from the fully scoped name of the CORBA interface on which the exception is raised. A structure is defined for each User Exception, which contains the body of information to be returned as part of that exception. The name of the structure follows the naming conventions used to map CORBA structure definitions.

Each User Exception that can be raised by an operation defined for a CORBA interface is mapped into an operation on the Exception interface. The name of the operation is constructed by prefixing the name of the exception with the string “_get_”. Each accessor operation defined takes one output parameter in which to return the body of information defined for the User Exception. The data type of the output parameter is a structure that is defined for the exception. The operation is defined to return an HRESULT value.

If a CORBA User Exception is to be raised, the value of the HRESULT returned to the caller is E_FAIL.

If the caller specified a non-NULL value for the Exceptions structure parameter, the callee must allocate the memory to hold the exception information and fill in the Exceptions structure as in Table 18-5.

Table 18-5 User Exceptions Structure

Member	Description
type	Indicates the type of CORBA exception that is being raised. Must be USER_EXCEPTION.
repositoryId	Indicates the repository identifier for the exception definition.
piUserException	Points to an interface with which to obtain information about the User Exception raised.

When data conversion errors occur while mapping the data types between object models (during a call from a COM client to a CORBA server), an HRESULT with the code E_DATA_CONVERSION and the facility value FACILITY_NULL is returned to the client.

Mapping User Exceptions: A Special Case

If a CORBA operation raises only one (COM_ERROR or COM_ERROREX) user exception (defined under Section 18.3.10.2, “Mapping for COM Errors,” on page 18-44), then the mapped COM operation should not have the additional parameter

for exceptions. This proviso enables a CORBA implementation of a preexisting COM interface to be mapped back to COM without altering the COM operation's original signature.

COM_ERROR (and COM_ERROREX) is defined as part of the CORBA to COM mapping. However, this special rule in effect means that a COM_ERROR raises clause can be added to an operation specifically to indicate that the operation was originally defined as a COM operation.

18.2.10.3 Mapping for Nested Types

OMG IDL and Microsoft MIDL/ODL do not agree on the scoping level of types declared within interfaces. Microsoft, for example, considers all types in a MIDL or ODL file to be declared at global scope. OMG IDL considers a type to be scoped within its enclosing module or interface. This means that to prevent accidental name collisions, types declared within OMG IDL modules and OMG IDL interfaces must be fully qualified in Microsoft IDL or ODL.

The OMG IDL construct:

```
Module BANK{
  interface ATM {
    enum type {CHECKS, CASH};
    Struct DepositRecord {
      string account;
      float amount;
      type kind;
    };
    void deposit (in DepositRecord val);
  };
};
```

Must be mapped in Microsoft MIDL as:

```
[uuid(...), object]
interface IBANK ATM : IUnknown {
  typedef [v1 enum] enum
    {BANK ATM CHECKS,
     BANK ATM CASH} BANK ATM type;
  typedef struct {
    LPSTR account;
    BANK ATM type kind;
  } BANK ATM DepositRecord;
  HRESULT deposit (in BANK ATM DepositRecord *val);
};
```

and to Microsoft ODL as:

```
[uuid(...)]
library BANK {
  ...
  [uuid(...), object]
```

```

interface IBANK ATM : IUnknown {
    typedef enum { BANK ATM CHECKS,
                  {BANK ATM CASH} BANK ATM type;
    typedef struct {
        LPSTR struct;
        float amount;
        BANK ATM type kind;
    } BANK ATM DepositRecord;
    HRESULT deposit (in BANK ATM DepositRecord *val);
};

```

18.2.10.4 Mapping for Operations

Operations defined for an interface are defined in OMG IDL within interface definitions. The definition of an operation constitutes the operations signature. An operation signature consists of the operation's name, parameters (if any), and return value. Optionally, OMG IDL allows the operation definition to indicate exceptions that can be raised, and the context to be passed to the object as implicit arguments, both of which are considered part of the operation.

OMG IDL parameter directional attributes **in**, **out**, **inout** map directly to Microsoft IDL and ODL parameter direction attributes [**in**], [**out**], [**in,out**]. Operation request parameters are represented as the values of **in** or **inout** parameters in OMG IDL, and operation response parameters are represented as the values of **inout** or **out** parameters. An operation return result can be any type that can be defined in OMG IDL, or void if a result is not returned.

The OMG IDL sample (shown below) illustrates the definition of two operations on the Bank interface. The names of the operations are bolded to make them stand out. Operations can return various types of data as results, including nothing at all. The operation **Bank::Transfer** is an example of an operation that does not return a value. The operation **Bank::Open Account** returns an object as a result of the operation.

```

// OMG IDL
#pragma ID::BANK::Bank"IDL:BANK/Bank:1,2"
interface Bank
{
    Account OpenAccount(    in float StartingBalance,
                           in AccountTypes Account(Type);
    void Transfer(         in Account Account1,
                           in Account Account2,
                           in float Account)
                           raises(InSufFunds);
};

```

The operations defined in the preceding OMG IDL code are mapped to the following lines of Microsoft IDL code:

```

// Microsoft IDL
[ object, uuid(682d22fb-78ac-0000-0c03-4d0000000000),
  pointer_default(unique) ]

```

```

interface IBANK Teller: IUnknown
{
    HRESULT OpenAccount(
        [in] float StartingBalance,
        [in] BANK_AccountTypes AccountType,
        [out] IBANK_Account ** ppiNewAccount );
    HRESULT Transfer(
        [in] IBANK_Account * Account1,
        [in] IBANK_Account * Account2,
        [in] float Amount,
        [out] BANK_TellerExceptions
            ** ppException);
};

```

and to the following statements in Microsoft ODL

```

// Microsoft ODL
[ uuid(682d22fb-78ac-0000-0c03-4d0000000000) odl ]
interface IBANK_Teller: IUnknown
{
    HRESULT OpenAccount(
        [in] float StartingBalance,
        [in] BANK_AccountTypes AccountType,
        [out, retval] IBANK_Account
            ** ppiNewAccount );
    HRESULT Transfer(
        [in] IBANK_Account * Account1,
        [in] IBANK_Account * Account2,
        [in] float Amount,
        [out]BANK_TellerExceptions
            ** ppException);
};

```

The ordering and names of parameters in the Microsoft IDL and ODL mapping is identical to the order in which parameters are specified in the text of the operation definition in OMG IDL. The COM mapping of all CORBA operations must obey the COM memory ownership and allocation rules specified.

It is important to note that the signature of the operation as written in OMG IDL is different from the signature of the same operation in Microsoft IDL or ODL. In particular, the result value returned by an operation defined in OMG IDL will be mapped as an output argument at the end of the signature when specified in Microsoft IDL or ODL. This allows the signature of the operation to be natural to the COM developer. When a result value is mapped as an output argument, the result value becomes an HRESULT. Without an HRESULT return value, there would be no way for COM to signal errors to clients when the client and server are not collocated. The value of the HRESULT is determined based on a mapping of the CORBA exception, if any, that was raised.

It is also important to note that if any user's exception information is defined for the operation, an additional parameter is added as the last argument of the operation signature. The user exception parameter follows the return value parameter, if both exist. See Section 18.2.10.2, "Mapping for exception types," on page 18-11 for further details.

18.2.10.5 *Mapping for Oneway Operations*

OMG IDL allows an operation's definition to indicate the invocation semantics the communication service must provide for an operation. This indication is done through the use of an operation attribute. Currently, the only operation attribute defined by CORBA is the oneway attribute.

The oneway attribute specifies that the invocation semantics are best-effort, which does not guarantee delivery of the request. Best-effort implies that the operation will be invoked, at most, once. Along with the invocation semantics, the use of the oneway operation attribute restricts an operation from having output parameters, must have no result value returned, and cannot raise any user-defined exceptions.

It may seem that the Microsoft IDL **maybe** operation attribute provides a closer match since the caller of an operation does not expect any response. However, Microsoft RPC maybe does not guarantee at most once semantics, and therefore is not sufficient. Because of this, the mapping of an operation defined in OMG IDL with the oneway operation attribute maps the same as an operation that has no output arguments.

18.2.10.6 *Mapping for Attributes*

OMG IDL allows the definition of attributes for an interface. Attributes are essentially a short-hand for a pair of accessor functions to an object's data; one to retrieve the value and possibly one to set the value of the attribute. The definition of an attribute must be contained within an interface definition and can indicate whether the value of the attribute can be modified or just read. In the example OMG IDL next, the attribute Profile is defined for the Customer interface and the read-only attribute is Balance-defined for the Account interface. The keyword attribute is used by OMG IDL to indicate that the statement is defining an attribute of an interface.

The definition of attributes in OMG IDL are restricted from raising any user-defined exceptions. Because of this, the implementation of an attribute's accessor function is limited to only raising system exceptions. The value of the HRESULT is determined based on a mapping of the CORBA exception, if any, that was raised.

```
// OMG IDL
struct CustomerData
{
    CustomerId Id;
    string      Name;
    string      SurName;
};
```

```
#pragma ID::BANK::Account "IDL:BANK/Account:3.1"
```

```
interface Account
```

```
{
    readonly attribute float Balance;
    float Deposit(in float amount) raises(InvalidAmount);
    float Withdrawal(in float amount) raises(InsufFunds, InvalidAmount);
    float Close( );
};
```

```
#pragma ID::BANK::Customer "IDL:BANK/Customer:1.2"
```

```
interface Customer
```

```
{
    attribute CustomerData Profile;
};
```

When mapping attribute statements in OMG IDL to Microsoft IDL or ODL, the name of the get accessor is the same as the name of the attribute prefixed with `_get_` in Microsoft IDL and contains the operation attribute `[propget]` in Microsoft ODL. The name of the put accessor is the same as the name of the attribute prefixed with `_put_` in Microsoft IDL and contains the operation attribute `[propput]` in Microsoft ODL.

Mapping for Read-Write Attributes

In OMG IDL, attributes are defined as supporting a pair of accessor functions: one to retrieve the value and one to set the value of the attribute, unless the keyword `readonly` precedes the attribute keyword. In the preceding example, the attribute `Profile` is mapped to the following statements in Microsoft IDL.

```
// Microsoft IDL
[ object, uuid(682d22fb-78ac-0000-0c03-4d0000000000),
  pointer_default(unique) ]
interface ICustomer : IUnknown
{
    HRESULT _get_Profile( [out] CustomerData * Profile );
    HRESULT _put_Profile( [in] CustomerData * Profile );
};
```

`Profile` is mapped to these statements in Microsoft ODL.

```
// Microsoft ODL
[ uuid(682d22fb-78ac-0000-0c03-4d0000000000) ]
interface IBANK_Customer : IUnknown
{
    [propget] HRESULT Profile(
        [out] BANK_CustomerData * val);
    [propput] HRESULT Profile(
        [in] BANK_CustomerData * val);
};
```

Note – The attribute is actually mapped as two different operations in both Microsoft IDL and ODL. The `IBANK_Customer::get_profile` operation (in Microsoft IDL) and the `[propget]` Profile operation (in Microsoft ODL) are used to retrieve the value of the attribute. The `IBANK_Customer::put_profile` operation is used to set the value of the attribute.

Mapping for Read-Only Attributes

In OMG IDL, an attribute preceded by the keyword `readonly` is interpreted as only supporting a single accessor function used to retrieve the value of the attribute. In the previous example, the mapping of the attribute `Balance` is mapped to the following statements in Microsoft IDL.

```
// Microsoft IDL
[ object, uuid(682d22fb-78ac-0000-0c03-4d0000000000) ]
interface IAccount: IUnknown
{
    HRESULT _get_Balance([out] float Balance);
};
```

and the following statements in Microsoft ODL.

```
// Microsoft ODL
[ uuid(682d22fb-78ac-0000-0c03-4d0000000000) ]
interface IAccount: IUnknown
{
    [propget] HRESULT Balance([out] float *val);
};
```

Note that only a single operation was defined since the attribute was defined to be read-only.

18.2.10.7 Indirection Levels for Operation Parameters

- For integral types (such as long, enum, char,...) these are passed by value as [in] parameters and by reference as out parameters.
- string/wstring parameters are passed as LPSTR/LPWSTR as an in parameter and LPSTR*/LPWSTR* as an out parameter.
- composite types (such as unions, structures, exceptions) are passed by reference for both [in] and [out] parameters.
- optional parameters are passed using double indirection (e.g., `IntfException ** val`).

18.2.11 Inheritance Mapping

Both CORBA and COM have similar models for individual interfaces. However, the models for inheritance and multiple interfaces are different.

In CORBA, an interface can singly or multiply inherit from other interfaces. In language bindings supporting typed object references, widening and narrowing support convert object references as allowed by the true type of that object.

However, there is no built-in mechanism in CORBA to access interfaces without an inheritance relationship. The run-time interfaces of an object, as defined in CORBA (for example, `CORBA::Object::is_a`, `CORBA::Object::get_interface`) use a description of the object's principle type, which is defined in OMG IDL. CORBA allows many ways in which implementations of interfaces can be structured, including using implementation inheritance.

In COM V2.0, interfaces can have single inheritance. However, as opposed to CORBA, there is a standard mechanism by which an object can have multiple interfaces (without an inheritance relationship between those interfaces) and by which clients can query for these at run-time. (It defines no common way to determine if two interface references refer to the same object, or to enumerate all the interfaces supported by an entity.)

An observation about COM is that some COM objects have a required minimum set of interfaces, which they must support. This type of statically defined interface relation is conceptually equivalent to multiple inheritance; however, discovering this relationship is only possible if ODL or type libraries are always available for an object.

COM describes two main implementation techniques: aggregation and delegation. C++ style implementation inheritance is not possible.

The mapping for CORBA interfaces into COM is more complicated than COM interfaces into CORBA, since CORBA interfaces might be multiply-inherited and COM does not support multiple interface inheritance.

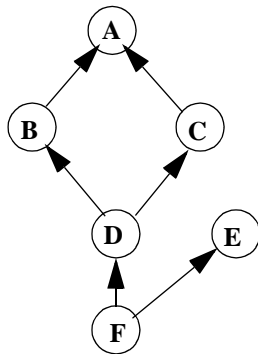
If a CORBA interface is singly inherited, this maps directly to single inheritance of interfaces in COM. The base interface for all CORBA inheritance trees is `IUnknown`. Note that the `Object` interface is not surfaced in COM. For single inheritance, although the most derived interface can be queried using `IUnknown::QueryInterface`, each individual interface in the inheritance hierarchy can also be queried separately.

The following rules apply to mapping CORBA to COM inheritance.

- Each OMG IDL interface that does not have a parent is mapped to an MIDL interface deriving from `IUnknown`.
- Each OMG IDL interface that inherits from a single parent interface is mapped to an MIDL interface that derives from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parent interfaces is mapped to an MIDL interface deriving from `IUnknown`.
- For each CORBA interface, the mapping for operations precede the mapping for attributes.
- Operations are sorted in ascending order based upon the ISO Latin-1 encoding values of the respective operation names.

- The resulting mapping of attributes within an interface are ordered based upon the attribute name. The attributes are similarly sorted in ascending order based upon the ISO-Latin-1 encoding values of the respective attribute names. If the attribute is not readonly, the get_<attribute name> method immediately precedes the set_<attribute name> method.

CORBA Interface Inheritance



COM Interface Inheritance

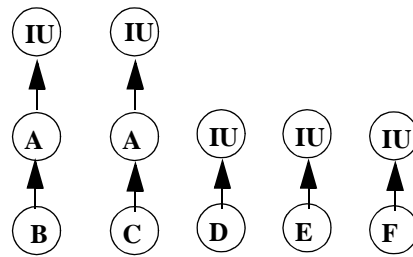


Figure 18-1 CORBA Interface Inheritance to COM Interface Inheritance Mapping

```

//OMG IDL
//
interface A {
    void opA();
    attribute long val;
};
interface B : A {
    void opB();
};
interface C : A {
    void opC();
};
interface D : B, C {
    void opD();
};
interface E {
    void opE();
};
interface F : D, E {
    void opF();
};

//Microsoft MIDL
//
[object, uuid(b97267fa-7855-e044-71fb-12fa8a4c516f)]
interface IA: IUnknown{
    HRESULT opA();
};
  
```

```

        HRESULT get_val([out] long * val);
        HRESULT set_val([in] long val);
};
[object, uuid(fa2452c3-88ed-1c0d-f4d2-fcf91ac4c8c6)]
interface IB: IA {
    HRESULT opB();
};
[object, uuid(dc3a6c32-f5a8-d1f8-f8e2-64566f815ed7)]
interface IC: IA {
    HRESULT opC();
};
[object, uuid(b718adec-73e0-4ce3-fc72-0dd11a06a308)]
interface ID: IUnknown {
    HRESULT opD();
};
[object, uuid(d2cb7bbc-0d23-f34c-7255-d924076e902f)]
interface IE: IUnknown{
    HRESULT opE();
};
[object, uuid(de6ee2b5-d856-295a-fd4d-5e3631fbfb93)]
interface IF: IUnknown {
    HRESULT opF();
};

```

Note that the **co-class** statement in Microsoft ODL allows the definition of an object class that allows QueryInterface between a set of interfaces.

Also note that when the interface defined in OMG IDL is mapped to its corresponding statements in Microsoft IDL, the name of the interface is preceded by the letter I to indicate that the name represents the name of an interface. This also makes the mapping more natural to the COM programmer, since the naming conventions used follow those suggested by Microsoft.

18.2.12 Mapping for Pseudo-Objects

CORBA defines a number of different kinds of pseudo-objects. Pseudo-objects differ from other objects in that they cannot be invoked with the Dynamic Invocation Interface (DII) and do not have object references. Most pseudo-objects cannot be used as general arguments. Currently, only the TypeCode and Principal pseudo-objects can be used as general arguments to a request in CORBA.

The CORBA NamedValue and NVList are not mapped into COM as arguments to COM operation signatures.

18.2.12.1 Mapping for TypeCode pseudo-object

CORBA TypeCodes represent the types of arguments or attributes and are typically retrieved from the interface repository. The mapping of the CORBA TypeCode interface follows the same rules as mapping any other CORBA interface to COM. The result of this mapping is as follows.

```

// Microsoft IDL or ODL
typedef struct { } TypeCodeBounds;
typedef struct { } TypeCodeBadKind;

[uuid(9556EA20-3889-11cf-9586-AA0004004A09), object,
 pointer_default(unique)]

interface ICORBA_TypeCodeUserExceptions : IUnknown
{
    HRESULT _get_Bounds( [out] TypeCodeBounds *pExceptionBody);
    HRESULT _get_BadKind( [out] TypeCodeBadKind * pExceptionBody );
};

typedef struct
{
    ExceptionType           type;
    LPTSTR                  repositoryId;
    long                    minorCode;
    CompletionStatus       completionStatus;
    ICORBA_SystemException * pSystemException;
    ICORBA_TypeCodeExceptions * pUserException;
} CORBA_TypeCodeExceptions;

typedef LPTSTR      RepositoryId;
typedef LPTSTR      Identifier;

typedef [vl_enum]
enum tagTCKind { tk_null = 0, tk_void, tk_short,
                tk_long, tk_ushort, tk_ulong,
                tk_float, tk_double, tk_octet,
                tk_any, tk_TypeCode,
                tk_principal, tk_objref,
                tk_struct, tk_union, tk_enum,
                tk_string, tk_sequence,
                tk_array, tk_alias, tk_except
} CORBA_TCKind;

[uuid(9556EA21-3889-11cf-9586-AA0004004A09), object,
 pointer_default(unique)]

interface ICORBA_TypeCode : IUnknown
{
    HRESULT equal(
        [in] ICORBA_TypeCode      * piTc,
        [out] boolean              * pbRetVal,
        [out] CORBA_TypeCodeExceptions** ppUserExceptions);
    HRESULT kind(
        [out] TCKind              * pRetVal,
        [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
    HRESULT id(
        [out] RepositoryId        * pszRetVal,
        [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
    HRESULT name(
        [out] Identifier          * pszRetVal,

```

```

        [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
HRESULT member_count(
    [out] unsigned long          * pulRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
HRESULT member_name(
    [in] unsigned long          ulIndex,
    [out] Identifier            * pszRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions );
HRESULT member_type(
    [in] unsigned long          ulIndex,
    [out] ICORBA_TypeCode      ** ppRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions );
HRESULT member_label(
    [in] unsigned long          ulIndex,
    [out] ICORBA_Any           ** ppRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions );
HRESULT discriminator_type(
    [out] ICORBA_TypeCode      ** ppRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions );
HRESULT default_index(
    [out] long                  * plRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
HRESULT length(
    [out] unsigned long          * pulRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
HRESULT content_type(
    [out] ICORBA_TypeCode      ** ppRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
HRESULT param_count(
    [out] long                  * plRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions);
HRESULT parameter(
    [in] long                    lIndex,
    [out] ICORBA_Any           ** ppRetVal,
    [out] CORBA_TypeCodeExceptions ** ppUserExceptions
);
}

```

Note – Use of the methods `param_count()` and `parameter()` is deprecated.

18.2.12.2 Mapping for context pseudo-object

This specification provides no mapping for CORBA's Context pseudo-object into COM. Implementations that choose to provide support for Context could do so in the following way. Context pseudo-objects should be accessed through the **ICORBA Context** interface. This would allow clients (if they are aware that the object they are dealing with is a CORBA object) to set a single Context pseudo-object to be used for all subsequent invocations on the CORBA object from the client process space until such time as the **ICORBA_Context** interface is released.

```

// Microsoft IDL and ODL
typedef struct

```

```

    {
        unsigned long cbMaxSize;
        unsigned long cbLengthUsed;
        [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
            LPTSTR * pszValue;
    } ContextPropertyValue;

[ object, uuid(74105F51-3C68-11cf-9588-AA0004004A09),
  pointer_default(unique) ]
interface ICORBA_Context: IUnknown
{
    HRESULT GetProperty( [in]LPTSTR Name,
                        [out] ContextPropertyValue
                        ** pValues );

    HRESULT SetProperty( [in] LPTSTR,
                        [in] ContextPropertyValue
                        * pValues);
};

```

If a COM client application knows it is using a CORBA object, the client application can use **QueryInterface** to obtain an interface pointer to the **ICORBA_Context** interface. Obtaining the interface pointer results in a CORBA context pseudo-object being created in the View, which is used with any CORBA request operation that requires a reference to a CORBA context object. The context pseudo-object should be destroyed when the reference count on the **ICORBA_Context** interface reaches zero.

This interface should only be generated for CORBA interfaces that have operations defined with the context clause.

18.2.12.3 Mapping for principal pseudo-object

The CORBA Principal is not currently mapped into COM. As both the COM and CORBA security mechanisms solidify, security interworking will need to be defined between the two object models.

18.2.13 Interface Repository Mapping

Name spaces within the CORBA interface repository are conceptually similar to COM type libraries. However, the CORBA interface repository looks, to the client, to be one unified service. Type libraries, on the other hand, are each stored in a separate file. Clients do not have a unified, hierarchical interface to type libraries.

Table 18-6 defines the mapping between equivalent CORBA and COM interface description concepts. Where there is no equivalent, the field is left blank.

Table 18-6 CORBA Interface Repository to OLE Type Library Mappings

TypeCode	TYPEDESC
Repository	
ModuleDef	ITypeLib
InterfaceDef	ITypeInfo
AttributeDef	VARDESC
OperationDef	FUNCDESC
ParameterDef	ELEMDESC
TypeDef	ITypeInfo
ConstantDef	VARDESC
ExceptionDef	

Using this mapping, implementations must provide the ability to call **Object::get_interface** on CORBA object references to COM objects to retrieve an InterfaceDef. When CORBA objects are accessed from COM, implementations may provide the ability to retrieve the ITypeInfo for a CORBA object interface using the IProvideClassInfo COM interface.

18.3 COM to CORBA Data Type Mapping

18.3.1 Mapping for Basic Data Types

The basic data types available in Microsoft IDL and ODL map to the corresponding data types available in OMG IDL as shown in Table 18-7.

Table 18-7 Microsoft IDL and ODL to OMG IDL Intrinsic Data Type Mappings

Microsoft IDL	Microsoft ODL	OMG IDL	Description
short	short	short	Signed integer with a range of $-2^{15} \dots 2^{15} - 1$
long	long	long	Signed integer with a range of $-2^{31} \dots 2^{31} - 1$
unsigned short	unsigned short	unsigned short	Unsigned integer with a range of $0 \dots 2^{16} - 1$
unsigned long	unsigned long	unsigned long	Unsigned integer with a range of $0 \dots 2^{32} - 1$
float	float	float	IEEE single -precision floating point number
double	double	double	IEEE double-precision floating point number

Table 18-7 Microsoft IDL and ODL to OMG IDL Intrinsic Data Type Mappings (Continued)

char	char	char	8-bit quantity limited to the ISO Latin-1 character set
boolean	boolean	boolean	8-bit quantity, which is limited to 1 and 0
byte	unsigned char	octet	8-bit opaque data type, guaranteed to not undergo any conversion during transfer between systems

18.3.2 Mapping for Constants

The mapping of the Microsoft IDL keyword `const` to OMG IDL `const` is almost exactly the same. The following Microsoft IDL definitions for constants:

```
// Microsoft IDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const float F = ...;
const double D = ...;
const char C = ...;
const boolean B = ...;
const string STR = "...";
```

map to the following OMG IDL definitions for constants.

```
// OMG IDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const float F = ...;
const double D = ...;
const char C = ...;
const boolean B = ...;
const string STR = "...";
```

18.3.3 Mapping for Enumerators

COM enumerations can have enumerators explicitly tagged with values. When COM enumerations are mapped into CORBA, the enumerators are presented in CORBA in increasing order according to their tagged values.

The Microsoft IDL or ODL specification:

```
// Microsoft IDL or ODL
typedef [v1_enum] enum tagA_or_B_orC { A = 0, B, C }
A_or_B_or_C;
```

would be represented as the following statements in OMG IDL:

```
// OMG IDL
enum A_or_B_or_C {B, C, A};
```

In this manner, the precedence relationship is maintained in the OMG system such that B is less than C is less than A.

OMG IDL does not support enumerators defined with explicit tagged values. The CORBA view of a COM object, therefore, is responsible for maintaining the correct tagged value of the mapped enumerators as they cross the view.

18.3.4 Mapping for String Types

COM support for strings includes the concepts of bounded and unbounded strings. Bounded strings are defined as strings that have a maximum length specified, whereas unbounded strings do not have a maximum length specified. COM also supports Unicode strings where the characters are wider than 8 bits. As in OMG IDL, non-Unicode strings in COM are NULL-terminated. The mapping of COM definitions for bounded and unbounded strings differs from that specified in OMG IDL.

Table 18-8 illustrates how to map the string data types in OMG IDL to their corresponding data types in both Microsoft IDL and ODL.

Table 18-8 Microsoft IDL/ODL to OMG IDL String Mappings

Microsoft IDL	Microsoft ODL	OMG IDL	Description
LPSTR [string,unique] char *	LPSTR,	string	Null-terminated 8-bit character string
BSTR	BSTR	wstring	Null-terminated 16-bit character string
LPWSTR [string,unique] char *	LPWSTR	wstring	Null-terminated Unicode string

If a COM Server returns a BSTR containing embedded nulls to a CORBA client, a `E_DATA_CONVERSION` exception will be raised.

18.3.4.1 Mapping for unbounded string types

The definition of an unbounded string in Microsoft IDL and ODL denotes the unbounded string as a stringified unique pointer to a character. The following Microsoft IDL statement

```
// Microsoft IDL
typedef [string, unique] char * UNBOUNDED_STRING;
```

is mapped to the following syntax in OMG IDL.

```
// OMG IDL
typedef string UNBOUNDED_STRING;
```

In other words, a value of type **UNBOUNDED_STRING** is a non-NULL pointer to a one-dimensional null-terminated character array whose extent and number of valid elements can vary at run-time.

18.3.4.2 Mapping for bounded string types

Bounded strings have a slightly different mapping between OMG IDL and Microsoft IDL. Bounded strings are expressed in Microsoft IDL as a “stringified nonconformant array.” The following Microsoft IDL and ODL definition for a bounded string:

```
// Microsoft IDL and ODL
const long N = ...;
typedef [string, unique] char (* BOUNDED_STRING) [N];
```

maps to the following syntax in OMG IDL.

```
// OMG IDL
const long N = ...;
typedef string<N> BOUNDED_STRING;
```

In other words, the encoding for a value of type **BOUNDED_STRING** is that of a null-terminated array of characters whose extent is known at compile time, and whose number of valid characters can vary at run-time.

18.3.4.3 Mapping for Unicode Unbounded String Types

The mapping for a Unicode unbounded string type in Microsoft IDL or ODL is no different from that used for ANSI string types. The following Microsoft IDL and ODL statement

```
// Microsoft IDL and ODL
typedef [string, unique] LPWSTR
UNBOUNDED_UNICODE_STRING;
```

is mapped to the following syntax in OMG IDL.

```
// OMG IDL
typedef wstring UNBOUNDED_UNICODE_STRING;
```

It is the responsibility of the mapping implementation to perform the conversions between ANSI and Unicode formats when dealing with strings.

18.3.4.4 Mapping for unicode bound string types

The mapping for a Unicode bounded string type in Microsoft IDL or ODL is no different from that used for ANSI string types. The following Microsoft IDL and ODL statements

```
// Microsoft IDL and ODL
const long N = ...;
typedef [string, unique] wchar t(*
BOUNDED_UNICODE_STRING) [N];
```

map to the following syntax in OMG IDL.

```
// OMG IDL
const long N = ...;
typedef wstring<N> BOUNDED_UNICODE_STRING;
```

It is the responsibility of the mapping implementation to perform the conversions between ANSI and Unicode formats when dealing with strings.

18.3.5 Mapping for Structure Types

Support for structures in Microsoft IDL and ODL maps bidirectionally to OMG IDL. Each structure member is mapped according to the mapping rules for that data type. The structure definition in Microsoft IDL or ODL is as follows.

```
// Microsoft IDL and ODL
typedef ... T0;
typedef ... T1;
...
typedef ...TN;
typedef struct
{
    T0 m0;
    T1 m1;
    ...
    TN mN;
} STRUCTURE;
```

The structure has an equivalent mapping in OMG IDL, as follows:

```

// OMG IDL
typedef ... T0
typedef ... T1;
...
typedef ... TN;
struct STRUCTURE
{
    T0 m0;
    T1 m1;
    ...
    Tn mn;
};

```

18.3.6 Mapping for Union Types

ODL unions are not discriminated unions and must be custom marshaled in any interfaces that use them. For this reason, this specification does not provide any mapping for ODL unions to CORBA unions.

MIDL unions, while always discriminated, are not required to be encapsulated. The discriminator for a nonencapsulated MIDL union could, for example, be another argument to the operation. The discriminants for MIDL unions are not required to be constant expressions.

18.3.6.1 Mapping for Encapsulated Unions

When mapping from Microsoft IDL to OMG IDL, Microsoft IDL encapsulated unions having constant discriminators are mapped to OMG IDL unions as shown next.

```

// Microsoft IDL
typedef enum
{
    dChar,
    dShort,
    dLong,
    dFloat,
    dDouble
} UNION_DISCRIMINATOR;

typedef union switch (UNION_DISCRIMINATOR _d)
{
    case dChar: char c;
    case dShort: short s;
    case dLong: long l;
    case dFloat: float f;
    case dDouble: double d;
    default: byte v[8];
}UNION_OF_CHAR_AND_ARITHMETIC;

```

The OMG IDL definition is as follows.

```

// OMG IDL
enum UNION_DISCRIMINATOR
{
    dChar,
    dShort,
    dLong,
    dFloat,
    dDouble
};

union UNION_OF_CHAR_AND_ARITHMETIC
switch(UNION_DISCRIMINATOR)
{
    case dChar: char c;
    case dShort: short s;
    case dLong: long l;
    case dFloat: float f;
    case dDouble: double d;
    default: octet v[8];
};

```

18.3.6.2 Mapping for nonencapsulated unions

Microsoft IDL nonencapsulated unions and Microsoft IDL encapsulated unions with nonconstant discriminators are mapped to an **any** in OMG IDL. The type of the **any** is determined at run-time during conversion of the Microsoft IDL union.

```

// Microsoft IDL
typedef [switch_type( short )] union
tagUNION_OF_CHAR_AND_ARITHMETIC
{
    [case(0)] char c;
    [case(1)] short s;
    [case(2)] long l;
    [case(3)] float f;
    [case(4)] double d;
    [default] byte v[8];
} UNION_OF_CHAR_AND_ARITHMETIC;

```

The corresponding OMG IDL syntax is as follows.

```

// OMG IDL
typedef any UNION_OF_CHAR_AND_ARITHMETIC;

```

18.3.7 Mapping for Array Types

COM supports fixed-length arrays, just as in CORBA. As in the mapping from OMG IDL to Microsoft IDL, the arrays can be mapped bidirectionally. The type of the array elements is mapped according to the data type mapping rules. The following statements in Microsoft IDL and ODL describe a nonconformant and nonvarying array of U.

```
// Microsoft IDL for T
const long N = ...;
typedef ... U;
typedef U ARRAY_OF_N[N];
typedef float DTYPE[0..10]; // Equivalent to [11]
```

The value N can be of any integral type, and const means (as in OMG IDL) that the value of N is fixed and known at compilation time. The generalization to multidimensional arrays follows the obvious trivial mapping of syntax.

The corresponding OMG IDL syntax is as follows.

```
// OMG IDL for T
const long N = ...;
typedef ... T;
typedef T ARRAY_OF_N[N];
typedef float DTYPE[11];
```

18.3.7.1 Mapping for nonfixed arrays

In addition to fixed length arrays, as well as conformant arrays, COM supports varying arrays, and conformant varying arrays. These are arrays whose bounds and size can be determined at run-time. Nonfixed length arrays in Microsoft IDL and ODL are mapped to sequence in OMG IDL, as shown in the following statements.

```
// Microsoft IDL
typedef short BTYPE[]; // Equivalent to [*];
typedef char CTYPE[*];
```

The corresponding OMG IDL syntax is as follows.

```
// OMG IDL
typedef sequence<short> BTYPE;
typedef sequence<char> CTYPE;
```

18.3.7.2 Mapping for SAFEARRAY

Microsoft ODL also defines SAFEARRAY as a variable length, variable dimension array. Both the number of dimensions and the bounds of the dimensions are determined at run-time. Only the element type is predefined. A SAFEARRAY in Microsoft ODL is mapped to a CORBA sequence, as shown in the following statements.

```
// Microsoft ODL
SAFEARRAY(element-type) * ArrayName;
```

```
// OMG IDL
typedef sequence<element-type> SequenceName;
```

If a COM server returns a multidimensional SAFEARRAY to a CORBA client, an E_DATA_CONVERSION exception will be raised.

18.3.8 Mapping for VARIANT

The COM VARIANT provides semantically similar functionality to the CORBA **any**. However, its allowable set of data types are currently limited to the data types supported by Automation. VARTYPE is an enumeration type used in the VARIANT structure. The structure member *vt* is defined using the data type VARTYPE. Its value acts as the discriminator for the embedded union and governs the interpretation of the union. The list of valid values for the data type VARTYPE are listed in Table 18-9, along with a description of how to use them to represent the OMG IDL **any** data type.

Table 18-9 Valid OLE VARIANT Data Types

Value	Description
VT_EMPTY	No value was specified. If an argument is left blank, you should not return VT_EMPTY for the argument. Instead, you should return the VT_ERROR value: DISP_E_MEMBERNOTFOUND.
VT_EMPTY VT_BYREF	Illegal.
VT_UI1	An unsigned 1-byte character is stored in <i>bVal</i> .
VT_UI1 VT_BYREF	A reference to an unsigned 1-byte character was passed; a pointer to the value is in <i>pbVal</i> .
VT_I2	A 2-byte integer value is stored in <i>iVal</i> .
VT_I2 VT_BYREF	A reference to a 2-byte integer was passed; a pointer to the value is in <i>piVal</i> .
VT_I4	A 4-byte integer value is stored in <i>lVal</i> .
VT_I4 VT_BYREF	A reference to a 4-byte integer was passed; a pointer to the value is in <i>plVal</i> .
VT_R4	An IEEE 4-byte real value is stored in <i>fltVal</i> .
VT_R4 VT_BYREF	A reference to an IEEE 4-byte real was passed; a pointer to the value is in <i>pfltVal</i> .
VT_R8	An 8-byte IEEE real value is stored in <i>dblVal</i> .
VT_R8 VT_BYREF	A reference to an 8-byte IEEE real was passed; a pointer to its value is in <i>pdblVal</i> .

Table 18-9 Valid OLE VARIANT Data Types (Continued)

VT_CY	A currency value was specified. A currency number is stored as an 8-byte, two's complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. The value is in <i>cyVal</i> .
VT_CY VT_BYREF	A reference to a currency value was passed; a pointer to the value is in <i>pcyVal</i> .
VT_BSTR	A string was passed; it is stored in <i>bstrVal</i> . This pointer must be obtained and freed via the BSTR functions.
VT_BSTR VT_BYREF	A reference to a string was passed. A BSTR*, which points to a BSTR, is in <i>pbstrVal</i> . The referenced pointer must be obtained or freed via the BSTR functions.
VT_NULL	A propagating NULL value was specified. This should not be confused with the NULL pointer. The NULL value is used for tri-state logic as with SQL.
VT_NULL VT_BYREF	Illegal.
VT_ERROR	An SCODE was specified. The type of error is specified in <i>code</i> . Generally, operations on error values should raise an exception or propagate the error to the return value, as appropriate.
VT_ERROR VT_BYREF	A reference to an SCODE was passed. A pointer to the value is in <i>pscode</i> .
VT_BOOL	A Boolean (True/False) value was specified. A value of 0xFFFF (all bits one) indicates True; a value of 0 (all bits zero) indicates False. No other values are legal.
VT_BOOL VT_BYREF	A reference to a Boolean value. A pointer to the Boolean value is in <i>pbool</i> .
VT_DATE	A value denoting a date and time was specified. Dates are represented as double-precision numbers, where midnight, January 1, 1900 is 2.0, January 2, 1900 is 3.0, and so on. The value is passed in <i>date</i> . This is the same numbering system used by most spreadsheet programs, although some incorrectly believe that February 29, 1900 existed, and thus set January 1, 1900 to 1.0. The date can be converted to and from an MS-DOS representation using VariantTimeToDosDateTime.
VT_DATE VT_BYREF	A reference to a date was passed. A pointer to the value is in <i>pdate</i> .

Table 18-9 Valid OLE VARIANT Data Types (Continued)

VT_DISPATCH	A pointer to an object was specified. The pointer is in <i>pdispVal</i> . This object is only known to implement IDispatch; the object can be queried as to whether it supports any other desired interface by calling QueryInterface on the object. Objects that do not implement IDispatch should be passed using VT_UNKNOWN.
VT_DISPATCH VT_BYREF	A pointer to a pointer to an object was specified. The pointer to the object is stored in the location referred to by <i>ppdispVal</i> .
VT_VARIANT	Illegal. VARIANTARGs must be passed by reference.
VT_VARIANT VT_BYREF	A pointer to another VARIANTARG is passed in <i>pvarVal</i> . This referenced VARIANTARG will never have the VT_BYREF bit set in <i>vt</i> , so only one level of indirection can ever be present. This value can be used to support languages that allow functions to change the types of variables passed by reference.
VT_UNKNOWN	A pointer to an object that implements the IUnknown interface is passed in <i>punkVal</i> .
VT_UNKNOWN VT_BYREF	A pointer to a pointer to the IUnknown interface is passed in <i>ppunkVal</i> . The pointer to the interface is stored in the location referred to by <i>ppunkVal</i> .
VT_ARRAY <anything>	An array of data type <anything> was passed. (VT_EMPTY and VT_NULL are illegal types to combine with VT_ARRAY.) The pointer in <i>pByrefVal</i> points to an array descriptor, which describes the dimensions, size, and in-memory location of the array. The array descriptor is never accessed directly, but instead is read and modified using functions.

A COM VARIANT is mapped to the CORBA **any** without loss. If at run-time a CORBA client passes an inconvertible **any** to a COM server, a DATA_CONVERSION exception is raised.

18.3.9 Mapping for Pointers

MIDL supports three types of pointers:

- Reference pointer; a non-null pointer to a single item. The pointer cannot represent a data structure with cycles or aliasing (two pointers to the same address).
- Unique pointer; a (possibly null) pointer to a single item. The pointer cannot represent a data structure with cycles or aliasing.
- Full pointer; a (possibly null) pointer to a single item. Full pointers can be used for data structures, which form cycles or have aliases.

A reference pointer is mapped to a CORBA sequence containing one element. Unique pointers and full pointers with no aliases or cycles are mapped to a CORBA sequence containing zero or one elements. If at run-time a COM client passes a full pointer containing aliases or cycles to a CORBA server, `E_DATA_CONVERSION` is returned to the COM client. If a COM server attempts to return a full pointer containing aliases or cycles to a CORBA client, a `DATA_CONVERSION` exception is raised.

18.3.10 Interface Mapping

COM is a binary standard based upon standard machine calling conventions. Although interfaces can be expressed in Microsoft IDL, Microsoft ODL, or C++, the following interface mappings between COM and CORBA will use Microsoft ODL as the language of expression for COM constructs.

COM interface pointers bidirectionally map to CORBA Object references with the appropriate mapping of Microsoft IDL and ODL interfaces to OMG IDL interfaces.

18.3.10.1 Mapping for Interface Identifiers

Interface identifiers are used in both CORBA and COM to uniquely identify interfaces. These allow the client code to retrieve information about, or to inquire about other interfaces of an object.

COM identifies interfaces using a structure similar to the DCE UUID (in fact, identical to a DCE UUID on Win32) known as an IID. As with CORBA, COM specifies that the textual names of interfaces are only for convenience and need not be globally unique.

The COM interface identifier (IID and CLSID) are bidirectionally mapped to the CORBA RepositoryId.

18.3.10.2 Mapping for COM Errors

COM will provide error information to clients only if an operation uses a return result of type `HRESULT`. The COM `HRESULT`, if zero, indicates success. The `HRESULT`, if nonzero, can be converted into an `SCODE` (the `SCODE` is explicitly specified as being the same as the `HRESULT` on Win32). The `SCODE` can then be examined to determine whether the call succeeded or failed. The error or success code, also contained within the `SCODE`, is composed of a “facility” major code (13 bits on Win32 and 4 bits on Win16) and a 16-bit minor code.

COM object developers are expected to use one of the predefined `SCODE` values, or use the facility `FACILITY_ITF` and an interface-specific minor code. `SCODE` values can indicate either success codes or error codes. A typical use is to overload the `SCODE` with a boolean value, using `S_OK` and `S_FALSE` success codes to indicate a true or false return. If the COM server returns `S_OK` or `S_FALSE`, a CORBA exception will not be raised and the value of the `SCODE` will be mapped as the return value. This is because COM operations, which are defined to return an `HRESULT`, are mapped to CORBA as returning an `HRESULT`.

Unlike CORBA, COM provides no standard way to return user-defined exception data to the client. Also, there is no standard mechanism in COM to specify the completion status of an invocation. In addition, it is not possible to predetermine what set of errors a COM interface might return. Although the set of success codes that can be returned from a COM operation must be fixed when the operation is defined, there is currently no machine-readable way to discover what the set of valid success codes are.

COM exceptions have a straightforward mapping into CORBA. COM system error codes are mapped to the CORBA standard exceptions. COM user-defined error codes are mapped to CORBA user exceptions.

COM system error codes are defined with the FACILITY_NULL and FACILITY_RPC facility codes. All FACILITY_NULL and FACILITY_RPC COM errors are mapped to CORBA standard exceptions. Table 18-10 lists the mapping from COM FACILITY_NULL exceptions to CORBA standard exceptions.

Table 18-10 Mapping from COM FACILITY_NULL Error Codes to CORBA Standard (System) Exceptions

COM	CORBA
E_OUTOFMEMORY	NO_MEMORY
E_INVALIDARG	BAD_PARAM
E_NOTIMPL	NO_IMPLEMENT
E_FAIL	UNKNOWN
E_ACCESSDENIED	NO_PERMISSION
E_UNEXPECTED	UNKNOWN
E_ABORT	UNKNOWN
E_POINTER	BAD_PARAM
E_HANDLE	BAD_PARAM

Table 18-11 lists the mapping from COM FACILITY_RPC exceptions to CORBA standard exceptions. All FACILITY_RPC exceptions not listed in this table are mapped to the new CORBA standard exception COM.

Table 18-11 Mapping from COM FACILITY_RPC Error Codes to CORBA Standard (System) Exceptions

COM	CORBA
RPC_E_CALL_CANCELED	TRANSIENT
RPC_E_CANTPOST_INSENDCALL	COMM_FAILURE
RPC_E_CANTCALLOUT_INEXTERNALCALL	COMM_FAILURE
RPC_E_CONNECTION_TERMINATED	NV_OBJREF

Table 18-11 Mapping from COM FACILITY_RPC Error Codes to CORBA Standard (System) Exceptions (Continued)

RPC_E_SERVER_DIED	INV_OBJREF
RPC_E_SERVER_DIED_DNE	INV_OBJREF
RPC_E_INVALID_DATAPACKET	COMM_FAILURE
RPC_E_CANTTRANSMIT_CALL	TRANSIENT
RPC_E_CLIENT_CANTMARSHAL_DATA	MARSHAL
RPC_E_CLIENT_CANTUNMARSHAL_DATA	MARSHAL
RPC_E_SERVER_CANTMARSHAL_DATA	MARSHAL
RPC_E_SERVER_CANTUNMARSHAL_DATA	MARSHAL
RPC_E_INVALID_DATA	COMM_FAILURE
RPC_E_INVALID_PARAMETER	BAD_PARAM
RPC_E_CANTCALLOUT_AGAIN	COMM_FAILURE
RPC_E_SYS_CALL_FAILED	NO_RESOURCES
RPC_E_OUT_OF_RESOURCES	NO_RESOURCES
RPC_E_NOT_REGISTERED	NO_IMPLEMENT
RPC_E_DISCONNECTED	INV_OBJREF
RPC_E_RETRY	TRANSIENT
RPC_E_SERVERCALL_REJECTED	TRANSIENT
RPC_E_NOT_REGISTERED	NO_IMPLEMENT

COM SCODEs, other than those previously listed, are mapped into CORBA user exceptions and will require the use of the raises clause in OMG IDL. Since the OMG IDL mapping from the Microsoft IDL and ODL is likely to be generated, this is not a burden to the average programmer. The following OMG IDL illustrates such a user exception.

```
// OMG IDL
exception COM_ERROREX
{
    long hresult;
    Any info;
};
```

The **COM_ERROREX** extension is designed to allow exposure of exceptions passed using the per-thread ErrorObject. The Any contained in the **COM_ERROREX** is defined to hold a CORBA object reference that supports the OMG IDL mapping for the **IErrorInfo** interface.

18.3.10.3 Mapping of Nested Data Types

Microsoft MIDL (and ODL) consider all definitions to be at global (or library) scope regardless of position in the file. This can lead to name collisions in datatypes across interfaces. Operations or types later in the file can refer to a datatype without fully qualifying the name even if the type is nested within another interface.

For purposes of mapping MIDL/ODL to OMG IDL, we treat nested datatypes as if they had been prepended with the name of the scoping level. Thus:

```
interface IA : IUnknown
{
    typedef enum {ONE, TWO, THREE} Count;
    HRESULT f([in] Count val);
}
```

is mapped as if it were defined as:

```
typedef enum {A_ONE, A_TWO, A_THREE} A_Count;
interface IA : IUnknown
{
    HRESULT f([in] A_Count val);
}
```

18.3.10.4 Mapping of Names

Microsoft MIDL and ODL support prefixing types/names with leading underscores. When mapping from Microsoft MIDL or ODL to OMG IDL, the leading underscores are removed.

Note – This simple rule is not sufficient to avoid all name collisions (such as MIDL types that clash with OMG IDL reserved names or situations where two operation names differ only in the leading underscore). However, this rule will cover many common cases and leads to a more natural mapping than prepending a character before the underscore.

18.3.10.5 Mapping for Operations

Operations defined for an interface are defined in Microsoft IDL and ODL within interface definitions. The definition of an operation constitutes the operations signature. An operation signature consists of the operation's name, parameters (if any), and return value. Unlike OMG IDL, Microsoft IDL and ODL does not allow the operation definition to indicate the error information that can be returned.

Microsoft IDL and ODL parameter directional attributes ([in], [out], [in, out]) map directly to OMG IDL (**in**, **out**, **inout**). Operation request parameters are represented as the values of [in] or [inout] parameters in Microsoft IDL, and operation response parameters are represented as the values of [inout] or [out] parameters. An

operation return result can be any type that can be defined in Microsoft IDL/ODL, or void if a result is not returned. By convention, most operations are defined to return an HRESULT. This provides a consistent way to return operation status information.

When Microsoft ODL methods are mapped to OMG IDL, they undergo the following transformations. First, if the last parameter is tagged with the Microsoft ODL keyword **retval**, that argument will be used as the return type of the operation. If the last parameter is not tagged with **retval**, then the signature is mapped directly to OMG IDL following the mapping rules for the data types of the arguments. Some example mappings from COM methods to OMG IDL operations are shown in the following code.

```
// Microsoft ODL
interface IFoo: IUnknown
{
    HRESULT stringify(    [in] VARIANT value,
                        [out, retval] LPSTR * pszValue);

    HRESULT permute(    [inout] short * value);

    HRESULT tryPermute([inout] short * value,
                      [out] long newValue);
};
```

In OMG IDL this becomes:

```
typedef long HRESULT;
interface IFoo: CORBA::Composite, CosLifeCycle::LifeCycleObject
{
    string stringify(in any value) raises (COM_ERROR),
    COM_ERROREX);
    HRESULT permute(inout short value);

    HRESULT tryPermute(inout short value, out long newValue)
};
```

18.3.10.6 Mapping for Properties

In COM, only Microsoft ODL and OLE Type Libraries provide support for describing properties. Microsoft IDL does not support this capability. Any operations that can be determined to be either a put/set or get accessor are mapped to an attribute in OMG IDL. Because Microsoft IDL does not provide a means to indicate that something is a property, a mapping from Microsoft IDL to OMG IDL will not contain mappings to the attribute statement in OMG IDL.

When mapping between Microsoft ODL or OLE Type Libraries, properties in COM are mapped in a similar fashion to that used to map attributes in OMG IDL to COM. For example, the following Microsoft ODL statements define the attribute Profile for

the ICustomer interface and the read-only attribute Balance for the IAccount interface. The keywords **[propput]** and **[propget]** are used by Microsoft ODL to indicate that the statement is defining a property of an interface.

```
// Microsoft ODL
interface IAccount
{
    [propget] HRESULT Balance([out, retval] float
                             * pfBalance );
    ...
};

interface ICustomer
{
    [propget] HRESULT Profile([out] CustomerData * Profile);
    [propput] HRESULT Profile([in] CustomerData * Profile);
};
```

The definition of attributes in OMG IDL are restricted from raising any user-defined exceptions. Because of this, the implementation of an attribute's accessor function is limited to raising system exceptions. The value of the HRESULT is determined by a mapping of the CORBA exception, if any, that was raised.

18.3.11 Mapping for Read-Only Attributes

In Microsoft ODL, an attribute preceded by the keyword **[propget]** is interpreted as only supporting an accessor function, which is used to retrieve the value of the attribute. In the example above, the mapping of the attribute Balance is mapped to the following statements in OMG IDL.

```
// OMG IDL
interface Account
{
    readonly attribute float Balance;
    ...
};
```

18.3.12 Mapping for Read-Write Attributes

In Microsoft ODL, an attribute preceded by the keyword **[propput]** is interpreted as only supporting an accessor function that is used to set the value of the attribute. In the previous example, the attribute Profile is mapped to the following statements in OMG IDL.

```
// OMG IDL
struct CustomerData
{
    CustomerId Id;
    string Name;
```

```
    string SurName;
};

interface Customer
{
    attribute CustomerData Profile;
    ...
};
```

Since CORBA does not have the concept of write-only attributes, the mapping must assume that a property that has the keyword [**propput**] is mapped to a single read-write attribute, even if there is no associated [**propget**] method defined.

18.3.12.1 *Inheritance Mapping*

Both CORBA and COM have similar models for individual interfaces. However, the models for inheritance and multiple interfaces are different.

In CORBA, an interface can singly or multiply inherit from other interfaces, and in language bindings supporting typed object references, widening and narrowing support convert object references as allowed by the true type of that object.

However, there is no built-in mechanism in CORBA to access interfaces without an inheritance relationship. The run-time interfaces of an object (for example, **CORBA::Object::is_a**, **CORBA::Object::get_interface**) use a description of the object's principle type, which is defined in OMG IDL. In terms of implementation, CORBA allows many ways in which implementations of interfaces can be structured, including using implementation inheritance.

In COM V2.0, interfaces can have single inheritance. However, as opposed to CORBA, there is a standard mechanism by which an object can have multiple interfaces (without an inheritance relationship between those interfaces) and by which clients can query for these at run-time. (It defines no common way to determine if two interface references refer to the same object, or to enumerate all the interfaces supported by an entity.)

An observation about COM is that some COM objects have a required minimum set of interfaces that they must support. This type of statically-defined interface relation is conceptually equivalent to multiple inheritance; however, discovering this relationship is only possible if ODL or type libraries are always available for an object.

COM describes two main implementation techniques: aggregation and delegation. C++ style implementation inheritance is not possible.

When COM interfaces are mapped into CORBA, their inheritance hierarchy (which can only consist of single inheritance) is directly mapped into the equivalent OMG IDL inheritance hierarchy.²

2. This mapping fails in some cases, for example, if operation names are the same.

Note that although it is possible, using Microsoft ODL to map multiple COM interfaces in a class to OMG IDL multiple inheritance, the necessary information is not available for interfaces defined in Microsoft IDL. As such, this specification does not define a multiple COM interface to OMG IDL multiple inheritance mapping. It is assumed that future versions of COM will merge Microsoft ODL and Microsoft IDL, at which time the mapping can be extended to allow for multiple COM interfaces to be mapped to OMG IDL multiple inheritance.

CORBA::Composite is a general-purpose interface used to provide a standard mechanism for accessing multiple interfaces from a client, even though those interfaces are not related by inheritance. Any existing ORB can support this interface, although in some cases a specialized implementation framework may be desired to take advantage of this interface.

```

module CORBA // PIDL
{
  interface Composite
  {
    Object query_interface(in RepositoryId whichOne);
  };
  interface Composable:Composite
  {
    Composite primary_interface();
  };
};

```

The root of a COM interface inheritance tree, when mapped to CORBA, is multiply-inherited from **CORBA::Composable** and **CosLifeCycle::LifeCycleObject**. Note that the **IUnknown** interface is not surfaced in OMG IDL. Any COM method parameters that require **IUnknown** interfaces as arguments are mapped, in OMG IDL, to object references of type **CORBA::Object**.

```

// Microsoft IDL or ODL
interface IFoo: IUnknown
{
  HRESULT inquire([in] IUnknown *obj);
};

```

In OMG IDL, this becomes:

```

interface IFoo: CORBA::Composable, CosLifeCycle::LifeCycleObject
{
  void inquire(in Object obj);
};

```

18.3.12.2 Type Library Mapping

Name spaces within the OLE Type Library are conceptually similar to CORBA interface repositories. However, the CORBA interface repository looks, to the client, to be one unified service. Type libraries, on the other hand, are each stored in a separate file. Clients do not have a unified, hierarchical interface to type libraries.

The following table defines the mapping between equivalent CORBA and COM interface description concepts. Where there is no equivalent, the field is left blank.

Table 18-12 CORBA Interface Repository to OLE Type Library Mappings

CORBA	COM
TypeCode	TYPEDESC
Repository	
ModuleDef	ITypeLib
InterfaceDef	ITypeInfo
AttributeDef	VARDESC
OperationDef	FUNCDESC
ParameterDef	ELEMDESC
TypeDef	ITypeInfo
ConstantDef	VARDESC
ExceptionDef	

Using this mapping, implementations must provide the ability to call **Object::get_interface** on CORBA object references to COM objects to retrieve an **InterfaceDef**. When CORBA objects are accessed from COM, implementations may provide the ability to retrieve the **ITypeInfo** for CORBA object interface using the **IProvideClassInfo** COM interface.