```
const ComponentId    TAG_DCE_SEC_MECH = 103; // Security Service
```

13.6.6.1 TAG_ORB_TYPE Component

It is often useful in the real world to be able to identify the particular kind of ORB an object reference is coming from, to work around problems with that particular ORB, or exploit shared efficiencies.

The **TAG_ORB_TYPE** component has an associated value of type **unsigned long**, encoded as a CDR encapsulation, designating an ORB type ID allocated by the OMG for the ORB type of the originating ORB. Anyone may register any ORB types by submitting a short (one-paragraph) description of the ORB type to the OMG, and will receive a new ORB type ID in return. A list of ORB type descriptions and values will be made available on the OMG web server.

The **TAG_ORB_TYPE** component can appear at most once in any IOR profile. For profiles supporting IIOP 1.1 or greater, it is optionally present.

13.6.6.2 TAG_ALTERNATE_IIOP_ADDRESS Component

In cases where the same object key is used for more than one internet location, the following standard IOR Component is defined for support in IIOP version 1.2.

The **TAG_ALTERNATE_IIOP_ADDRESS** component has an associated value of type

```
struct {
    string HostID,
    unsigned short Port
};
```

encoded as a CDR encapsulation.

Zero or more instances of the **TAG_ALTERNATE_IIOP_ADDRESS** component type may be included in a version 1.2 **TAG_INTERNET_IOP** Profile. Each of these alternative addresses may be used by the client orb, in addition to the host and port address expressed in the body of the Profile. In cases where one or more **TAG_ALTERNATE_IIOP_ADDRESS** components are present in a **TAG_INTERNET_IOP** Profile, no order of use is prescribed by Version 1.2 of IIOP.

13.6.6.3 Other Components

The following standard components are specified in various OMG specifications:

- **TAG_CODE_SETS** - See Section 13.10.2.4, "CodeSet Component of IOR Multi-Component Profile" on page 13-42.
- **TAG_POLICIES** - See CORBA Messaging - chapter 22.
- **TAG_SEC_NAME** - See the Security Service specification, Mechanism Tags section.

- **TAG_ASSOCIATION_OPTIONS** - See the Security Service specification, Tag Association Options section.
- **TAG_SSL_SEC_TRANS** - See the Security Service specification, Mechanism Tags section.
- **TAG_GENERIC_SEC_MECH** and all other tags with names in the form **TAG_*_SEC_MECH** - See the Security Service specification, Mechanism Tags section.
- **TAG_FIREWALL_SEC** - See the Firewall specification (orbos/98-05-04).
- **TAG_SCCP_CONTACT_INFO** - See the CORBA/IN Interworking specification (telecom/98-10-03).
- **TAG_JAVA_CODEBASE** - See the Java to IDL Language Mapping specification (formal/99-07-59), Codebase Transmission section.
- **TAG_TRANSACTION_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_MESSAGE_ROUTERS** - See CORBA Messaging (chapter 22).
- **TAG_OTS_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_INV_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_INET_SEC_TRANS** - See the Security Service specification (formal/00-06-25).
- **TAG_COMPLETE_OBJECT_KEY** (See Section 16.5.4, "Complete Object Key Component" on page 16-19).
- **TAG_ENDPOINT_ID_POSITION** (See Section 16.5.5, "Endpoint ID Position Component" on page 16-20).
- **TAG_LOCATION_POLICY** (See Section 16.5.6, "Location Policy Component" on page 16-20).
- **TAG_DCE_STRING_BINDING** (See Section 16.5.1, "DCE-CIOP String Binding Component" on page 16-17).
- **TAG_DCE_BINDING_NAME** (See Section 16.5.2, "DCE-CIOP Binding Name Component" on page 16-18).
- **TAG_DCE_NO_PIPES** (See Section 16.5.3, "DCE-CIOP No Pipes Component" on page 16-19).

13.6.7 Profile and Component Composition in IORs

The following rules augment the preceding discussion:

1. Profiles must be independent, complete, and self-contained. Their use shall not depend on information contained in another profile.
2. Any invocation uses information from exactly one profile.

3. Information used to drive multiple inter-ORB protocols may coexist within a single profile, possibly with some information (e.g., components) shared between the protocols, as specified by the specific protocols.

4. Unless otherwise specified in the definition of a particular profile, multiple profiles with the same profile tag may be included in an IOR.

5. Unless otherwise specified in the definition of a particular component, multiple components with the same component tag may be part of a given profile within an IOR.

6. A **TAG_MULTIPLE_COMPONENTS** profile may hold components shared between multiple protocols. Multiple such profiles may exist in an IOR.

7. The definition of each protocol using a **TAG_MULTIPLE_COMPONENTS** profile must specify which components it uses, and how it uses them.

8. Profile and component definitions can be either public or private. Public definitions are those whose tag and data format is specified in OMG documents. For private definitions, only the tag is registered with OMG.

9. Public component definitions shall state whether or not they are intended for use by protocols other than the one(s) for which they were originally defined, and dependencies on other components.

The OMG is responsible for allocating and registering protocol and component tags. Neither allocation nor registration indicates any "standard" status, only that the tag will not be confused with other tags. Requests to allocate tags should be sent to *tag_request@omg.org*.

13.6.8 IOR Creation and Scope

IORs are created from object references when required to cross some kind of referencing domain boundary. ORBs will implement object references in whatever form they find appropriate, including possibly using the IOR structure. Bridges will normally use IORs to mediate transfers where that standard is appropriate.

13.6.9 Stringified Object References

Object references can be "stringified" (turned into an external string form) by the **ORB::object_to_string** operation, and then "destringified" (turned back into a programming environment's object reference representation) using the **ORB::string_to_object** operation.

There can be a variety of reasons why being able to parse this string form might *not* help make an invocation on the original object reference:

- Identifiers embedded in the string form can belong to a different domain than the ORB attempting to destringify the object reference.
- The ORBs in question might not share a network protocol, or be connected.
- Security constraints may be placed on object reference destringification.

Nonetheless, there is utility in having a defined way for ORBs to generate and parse stringified IORs, so that in some cases an object reference stringified by one ORB could be destringified by another.

To allow a stringified object reference to be internalized by what may be a different ORB, a stringified IOR representation is specified. This representation instead establishes that ORBs could parse stringified object references using that format. This helps address the problem of bootstrapping, allowing programs to obtain and use object references, even from different ORBs.

The following is the representation of the stringified (externalized) IOR:

```
(1)    <oref>     ::= <prefix> <hex_Octets>
(2)    <prefix>   ::= <i><o><r>":"
(3)    <hex_Octets> ::= <hex_Octet> {<hex_Octet>}*
(4)    <hex_Octet> ::= <hexDigit> <hexDigit>
(5)    <hexDigit> ::= <digit> | <a> | <b> | <c> | <d> | <e> | <f>
(6)    <digit>    ::= "0" | "1" | "2" | "3" | "4" | "5" |
                     |  "6" | "7" | "8" | "9"
(7)    <a>        ::= "a" | "A"
(8)    <b>        ::= "b" | "B"
(9)    <c>        ::= "c" | "C"
(10)   <d>        ::= "d" | "D"
(11)   <e>        ::= "e" | "E"
(12)   <f>        ::= "f" | "F"
(13)   <i>        :: = "i" | "I"
(14)   <o>        :: = "o" | "O"
(15)   <r>        :: = "r" | "R"
```

Note – The case for characters in a stringified IOR is not significant.

The hexadecimal strings are generated by first turning an object reference into an IOR, and then encapsulating the IOR using the encoding rules of CDR, as specified in GIOP 1.0. (See Section 15.3, "CDR Transfer Syntax" on page 15-4 for more information.) The content of the encapsulated IOR is then turned into hexadecimal digit pairs, starting with the first octet in the encapsulation and going until the end. The high four bits of each octet are encoded as a hexadecimal digit, then the low four bits.

13.6.10 Object URLs

To address the problem of bootstrapping and allow for more convenient exchange of human-readable object references, **ORB::string_to_object** allows URLs in the **corbaloc** and **corbaname** formats to be converted into object references.

If conversion fails, string_to_object raises a BAD_PARAM exception with one of following standard minor codes, as appropriate:

- 7 - string_to_object conversion failed due to bad scheme name

- 8 - string_to_object conversion failed due to bad address
- 9 - string_to_object conversion failed due to bad bad schema specific part
- 10 - string_to_object conversion failed due to non specific reason

13.6.10.1 corbaloc URL

The **corbaloc** URL scheme provides stringified object references that are more easily manipulated by users than **IOR** URLs. Currently, **corbaloc** URLs denote objects that can be contacted by IIOP or **resolve_initial_references**. Other transport protocols can be explicitly specified when they become available. Examples of IIOP and **resolve_initial_references** (**rir:**) based **corbaloc** URLs are:

```
corbaloc::555xyz.com/Prod/TradingService
corbaloc:iiop:1.1@555xyz.com/Prod/TradingService
corbaloc::555xyz.com,:556xyz.com:80/Dev/NameService
corbaloc:rir:/TradingService
corbaloc:rir:/NameService
```

A **corbaloc** URL contains one or more:

- protocol identifiers
- protocol specific components such as address and protocol version information

When the **rir** protocol is used, no other protocols are allowed.

After the addressing information, a **corbaloc** URL ends with a single object key.

The full syntax is:

```
<corbaloc>          = "corbaloc:"<obj_addr_list>["/"<key_string>]
<obj_addr_list>     = [<obj_addr> ","]* <obj_addr>
<obj_addr>          = <prot_addr> | <future_prot_addr>
<prot_addr>         = <rir_prot_addr> | <iiop_prot_addr>

<rir_prot_addr>     = <rir_prot_token>":"
<rir_prot_token>    = "rir"

<iiop_prot_addr>    = <iiop_id><iiop_addr>
<iiop_id>           = ":" | <iiop_prot_token>":"
<iiop_prot_token>   = "iiop"
<iiop_addr>         = [<version> <host> [":" <port>]]
<host>              = DNS_style_Host_Name | ip_address
<version>           = <major> "." <minor> "@" | empty_string
<port>              = number
<major>             = number
<minor>             = number

<future_prot_addr>  = <future_prot_id><future_prot_addr>
<future_prot_id>    = <future_prot_token>":"
<future_prot_token> = possible examples: "atm" | "dce"
<future_prot_addr>  = protocol specific address
```

<key_string> = **<string> | empty_string**

Where:

obj_addr_list: comma-separated list of protocol id, version, and address information. This list is used in an implementation-defined manner to address the object An object may be contacted by any of the addresses and protocols.

Note – If the `rir` protocol is used, no other protocols are allowed.

---

obj_addr: A protocol identifier, version tag, and a protocol specific address. The comma ',' and '/' characters are specifically prohibited in this component of the URL.

rir_prot_addr: resolve_initial_references protocol identifier. This protocol does not have a version tag or address. See Section 13.6.10.2, "`corbaloc:rir` URL".

iiop_prot_addr: iiop protocol identifier, version tag, and address containing a DNS-style host name or IP address. See Section 13.6.10.3, "`corbaloc:iiop` URL"" for the iiop specific definitions.

future_prot_addr: a placeholder for future **corbaloc** protocols.

future_prot_id: token representing a protocol terminated with a ":".

future_prot_token: token representing a protocol. Currently only "**iiop**" and **"rir"** are defined.

future_prot_addr: a protocol specific address and possibly protocol version information. An example of this for **iiop** is "**1.1@555xyz.com**".

key_string: a stringified object key.

The **key_string** corresponds to the octet sequence in the **object_key** member of a GIOP **Request** or **LocateRequest** header as defined in section 15.4 of CORBA 2.3. The **key_string** uses the escape conventions described in RFC 2396 to map away from octet values that cannot directly be part of a URL. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

";" | "/" | ":" | "?"| ":" | "@" | "&" | "=" | "+" | "$" |
"," | "-" | "_" | "!" | "~" | "*" | "'" | "(" | ")"

The **key_string** is not NUL-terminated.

13.6.10.2 corbaloc:rir URL

The corbaloc:rir URL is defined to allow access to the ORB's configured initial references through a URL.

The protocol address syntax is:

<rir_prot_addr> = **<rir_prot_token>":"**
<rir_prot_token> = **"rir"**

Where:

rir_prot_addr: resolve_initial_references protocol identifier. There is no version or address information when **rir** is used.

rir_prot_token: The token "**rir**" identifies this protocol..

For a **corbaloc:rir** URL, the **<key_string>** is used as the argument to **resolve_initial_references**. An empty **<key_string>** is interpreted as the default "**NameService**".

The `rir` protocol can not be used with any other protocol in a URL.

13.6.10.3 corbaloc:iiop URL

The corbaloc:iiop URL is defined for use in TCP/IP- and DNS-centric environments The full protocol address syntax is:

```
<iiop_prot_addr>     = <iiop_id><iiop_addr>
<iiop_id>            = <iiop_default> | <iiop_prot_token>":"
<iiop_default>       = ":"
<iiop_prot_token>    = "iiop"
<iiop_addr>          = [<version> <host> [":" <port>]]
<host>               = DNS_style_Host_Name | ip_address
<version>            = <major> "." <minor> "@" | empty_string
<port>               = number
<major>              = number
<minor>              = number
```

Where:

iiop_prot_addr: iiop protocol identifier, version tag, and address containing a DNS-style host name or IP address.

iiop_id: tokens recognized to indicate an iiop protocol corbaloc.

iiop_default: default token indicating iiop protocol, ":".

iiop_prot_token: iiop protocol token, "**iiop**"

iiop_address: a single address

host: DNS-style host name or IP address. If not present, the local host is assumed.

version: a major and minor version number, separated by '.' and followed by '@'. If the version is absent, 1.0 is assumed.

ip_address: numeric IP address (dotted decimal notation)

port: port number the agent is listening on (see below). Default is *2809*.

13.6.10.4 corbaloc Server Implementation

The only requirements on an object advertised by a `corbaloc` URL are that there must be a software agent listening on the host and port specified by the URL. This agent must be capable of handling GIOP **Request** and **LocateRequest** messages targeted at the object key specified in the URL.

A normal CORBA server meets these criteria. It is also possible to implement lightweight object location forwarding agents that respond to GIOP `Request` messages with `Reply` messages with a **LOCATION_FORWARD** status, and respond to GIOP **LocateRequest** messages with **LocateReply** messages.

13.6.10.5 corbaname URL

The **corbaname** URL scheme is described in the Naming Service specification. It extends the capabilities of the **corbaloc** scheme to allow URLs to denote entries in a Naming Service. Resolving **corbaname** URLs does not require a Naming Service implementation in the ORB core. Some examples are:

corbaname::555objs.com#a/string/path/to/obj

This URL specifies that at host **555objs.com**, a object of type **NamingContext** (with an object key of **NameService**) can be found, or alternatively, that an agent is running at that location which will return a reference to a **NamingContext**. The (stringified) name **a/string/path/to/obj** is then used as the argument to a **resolve** operation on that `NamingContext`. The URL denotes the object reference that results from that lookup.

corbaname:rir:#a/local/obj

This URL specifies that the stringified name **a/local/obj** is to be resolved relative to the naming context returned by **resolve_initial_references("NameService")**.

13.6.10.6 Future corbaloc URL Protocols

This specification only defines use of iiop with corbaloc. New protocols can be added to **corbaloc** as required. Each new protocol must implement the <future_prot_addr> component of the URL and define a described in Section 13.6.10.1, "`corbaloc` URL" on page 13-24."

A possible example of a future corbaloc URL that incorporates an ATM address is:

corbaloc:iiop:xyz.com,atm:E.164:358.400.1234567/dev/test/objectX

13.6.10.7 Future URL Schemes

Several currently defined non-CORBA URL scheme names are reserved. Implementations may choose to provide these or other URL schemes to support additional ways of denoting objects with URLs.

Table 13-1 lists the required and some optional formats.

Table 13-1 URL formats

Scheme	Description	Status
IOR:	Standard stringified IOR format	Required
corbaloc:	Simple object reference. rir: must be supported.	Required
corbaname:	CosName URL	Required
file://	Specifies a file containing a URL/IOR	Optional
ftp://	Specifies a file containing a URL/IOR that is accessible via ftp protocol.	Optional
http://	Specifies an HTTP URL that returns an object URL/IOR.	Optional

13.7 Service Context

Emerging specifications for Object Services occasionally require service-specific context information to be passed implicitly with requests and replies. The Interoperability specifications define a mechanism for identifying and passing this service-specific context information as "hidden" parameters. The specification makes the following assumptions:

- Object Service specifications that need additional context passed will completely specify that context as an OMG IDL data type.
- ORB APIs will be provided that will allow services to supply and consume context information at appropriate points in the process of sending and receiving requests and replies.
- It is an ORB's responsibility to determine when to send service-specific context information, and what to do with such information in incoming messages. It may be possible, for example, for a server receiving a request to be unable to de-encapsulate and use a certain element of service-specific context, but nevertheless still be able to successfully reply to the message.

As shown in the following OMG IDL specification, the IOP module provides the mechanism for passing Object Service–specific information. It does not describe any service-specific information. It only describes a mechanism for transmitting it in the most general way possible. The mechanism is currently used by the DCE ESIOP and could also be used by the Internet Inter-ORB protocol (IIOP) General Inter_ORB Protocol (GIOP).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by OMG. Service context ID values are of type **unsigned long**. Object service specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

The marshaling of Object Service data is described by the following OMG IDL:

```
module IOP {        // IDL

    typedef unsigned long              ServiceId;

    struct ServiceContext {
        ServiceId           context_id;
        sequence <octet>    context_data;
    };
    typedef sequence <ServiceContext>ServiceContextList;
};
```

The context data for a particular service will be encoded as specified for its service-specific OMG IDL definition, and that encoded representation will be encapsulated in the **context_data** member of **IOP::ServiceContext**. (See Section 15.3.3, "Encapsulation" on page 15-14). The **context_id** member contains the service ID value identifying the service and data format. Context data is encapsulated in octet sequences to permit ORBs to handle context data without unmarshaling, and to handle unknown context data types.

During request and reply marshaling, ORBs will collect all service context data associated with the *Request* or *Reply* in a **ServiceContextList**, and include it in the generated messages. No ordering is specified for service context data within the list. The list is placed at the beginning of those messages to support security policies that may need to apply to the majority of the data in a request (including the message headers).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by the OMG. Service context ID values are of type unsigned long. Object service specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

The high-order 20 bits of service-context ID contain a 20-bit vendor service context codeset ID (VSCID); the low-order 12 bits contain the rest of the service context ID. A vendor (or group of vendors) who wish to define a specific set of service context IDs should obtain a unique VSCID from the OMG, and then define a specific set of service context IDs using the VSCID for the high-order bits.

The VSCID of zero is reserved for use for OMG-defined standard service context IDs (i.e., service context IDs in the range 0-4095 are reserved as OMG standard service contexts).

13.7.1 Standard Service Contexts

```
module IOP {        // IDL
    const ServiceId         TransactionService = 0;
    const ServiceId         CodeSets = 1;
    const ServiceId         ChainBypassCheck = 2;
    const ServiceId         ChainBypassInfo = 3;
    const ServiceId         LogicalThreadId = 4;
    const ServiceId         BI_DIR_IIOP = 5;
```

```
    const ServiceId     SendingContextRunTime = 6;
    const ServiceId     INVOCATION_POLICIES = 7;
    const ServiceId     FORWARDED_IDENTITY = 8;
    const ServiceId     UnknownExceptionInfo = 9;
    const ServiceId     RTCorbaPriority = 10;
    const ServiceId     RTCorbaPriorityRange = 11;
    const ServiceId     ExceptionDetailMessage = 14;
};
```

The standard ServiceIds currently defined are:

- **TransactionService** identifies a CDR encapsulation of the **CosTransactions::PropogationContext** defined in the Object Transaction Service specification (formal/00-06-28).
- **CodeSets** identifies a CDR encapsulation of the **CONV_FRAME::CodeSetContext** defined in Section 13.10.2.5, "GIOP Code Set Service Context" on page 13-43.
- DCOM-CORBA Interworking uses three service contexts as defined in "DCOM-CORBA Interworking" in the "Interoperability with non-CORBA Systems"chapter. They are:
  - **ChainBypassCheck**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassCheck**. This is carried only in a **Request** message as described in Section 20.9.1, "CORBA Chain Bypass" on page 20-19.
  - **ChainBypassInfo**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassInfo**. This is carried only in a **Reply** message as described in Section 20.9.1, "CORBA Chain Bypass" on page 20-19.
  - **LogicalThreadId**, which carries a CDR encapsulation of the **struct CosBridging::LogicalThreadId** as described in Section 20.10, "Thread Identification" on page 20-21.
- **BI_DIR_IIOP** identifies a CDR encapsulation of the **IIOP::BiDirIIOPServiceContext** defined in Section 15.8, "Bi-Directional GIOP" on page 15-55.
- **SendingContextRunTime** identifies a CDR encapsulation of the IOR of the **SendingContext::RunTime** object (see Section 5.6, "Access to the Sending Context Run Time" on page 5-18).
- For information on **INVOCATION_POLICIES** refer to CORBA Messaging (chapter 22).
- For information on **FORWARDED_IDENTITY** refer to the Firewall specification (orbos/98-05-04).
- **UnknownExceptionInfo** identifies a CDR encapsulation of a marshaled instance of a **java.lang.throwable** or one of its subclasses as described in Java to IDL Language Mapping, "Mapping of UnknownExceptionInfo Service Context," section.
- For information on **RTCorbaPriority** refer to the Real-time CORBA (chapter 24).

- For information on **RTCorbaPriorityRange** refer to the Real-time CORBA (chapter 24).
- **ExceptionDetailMessage** identifies a CDR encapsulation of a wstring, encoded using GIOP 1.2 with a TCS-W of UTF-16. This service context may be sent on Reply messages with a reply_status of SYSTEM_EXCEPTION or USER_EXCEPTION. The usage of this service context is defined by language mappings.

13.7.2 Service Context Processing Rules

Service context IDs are associated with a specific version of GIOP, but will always be allocated in the OMG service context range. This allows any ORB to recognize when it is receiving a standard service context, even if it has been defined in a version of GIOP that it does not support.

The following are the rules for processing a received service context:

- The service context is in the OMG defined range:
  - If it is valid for the supported GIOP version, then it must be processed correctly according to the rules associated with it for that GIOP version level.
  - If it is not valid for the GIOP version, then it may be ignored by the receiving ORB, however it must be passed on through a bridge and must be made available to interceptors. No exception shall be raised.
- The service context is not in the OMG-defined range:
  - The receiving ORB may choose to ignore it, or process it if it "understands" it, however the service context must be passed on through a bridge and must made available to interceptors.

13.8 Coder/Decoder Interfaces

The formats of IOR components and service context data used by ORB services are often defined as CDR encapsulations encoding instances of IDL defined data types. The **Codec** provides a mechanism to transfer these components between their IDL data types and their CDR encapsulation representations.

A **Codec** is obtained from the **CodecFactory**. The **CodecFactory** is obtained through a call to **ORB::resolve_initial_references ("CodecFactory")**.

13.8.1 Codec Interface

```
module IOP {
    local interface Codec {
        exception InvalidTypeForEncoding {};
        exception FormatMismatch {};
        exception TypeMismatch {};

        CORBA::OctetSeq encode (in any data)
```

```
            raises (InvalidTypeForEncoding);
        any decode (in CORBA::OctetSeq data)
            raises (FormatMismatch);
        CORBA::OctetSeq encode_value (in any data)
            raises (InvalidTypeForEncoding);
        any decode_value (
            in CORBA::OctetSeq data,
            in CORBA::TypeCode tc)
            raises (FormatMismatch, TypeMismatch);
    };
};
```

13.8.1.1 Exceptions

InvalidTypeForEncoding

This exception is raised by **encode** or **encode_value** when the type is invalid for the encoding. For example, this exception is raised if the encoding is **ENCODING_CDR_ENCAPS** version 1.0 and a type that does not exist in that version, such as wstring, is passed to the operation.

FormatMismatch

This exception is raised by **decode** or **decode_value** when the data in the octet sequence cannot be decoded into an any.

TypeMismatch

This exception is raised by **decode_value** when the given **TypeCode** does not match the given octet sequence.

13.8.1.2 Operations

encode

Convert the given **any** into an octet sequence based on the encoding format effective for this **Codec**.

This operation may raise **InvalidTypeForEncoding**.

Parameter

data	The data, in the form of an **any**, to be encoded into an octet sequence.

Return Value

An octet sequence containing the encoded **any**. This octet sequence contains both the **TypeCode** and the data of the type.

decode

Decode the given octet sequence into an **any** based on the encoding format effective for this **Codec**.

This operation raises **FormatMismatch** if the octet sequence cannot be decoded into an **any**.

Parameter

data	The data, in the form of an octet sequence, to be decoded into an **any**.

Return Value

An **any** containing the data from the decoded octet sequence.

encode_value

Convert the given any into an octet sequence based on the encoding format effective for this **Codec**. Only the data from the any is encoded, not the **TypeCode**.

This operation may raise **InvalidTypeForEncoding**.

Parameter

data	The data, in the form of an **any**, to be encoded into an octet sequence.

Return Value

An octet sequence containing the data from the encoded **any**.

decode_value

Decode the given octet sequence into an any based on the given **TypeCode** and the encoding format effective for this Codec.

This operation raises **FormatMismatch** if the octet sequence cannot be decoded into an any.

Parameter

data	The data, in the form of an octet sequence, to be decoded into an **any**.
tc	The **TypeCode** to be used to decode the data.

Return Value

An **any** containing the data from the decoded octet sequence.

13.8.2 Codec Factory

```
module IOP {
    typedef short EncodingFormat;
    const EncodingFormat ENCODING_CDR_ENCAPS = 0;
```

```
    struct Encoding {
        EncodingFormat format;
        octet major_version;
        octet minor_version;
    };

    local interface CodecFactory {
        exception UnknownEncoding {};
        Codec create_codec (in Encoding enc)
            raises (UnknownEncoding);
    };
};
```

13.8.2.1 Encoding Structure

The **Encoding** structure defines the encoding format of a **Codec**. It details the encoding format, such as CDR Encapsulation encoding, and the major and minor versions of that format.

The encodings which shall be supported are:

- **ENCODING_CDR_ENCAPS**, version 1.0;
- **ENCODING_CDR_ENCAPS**, version 1.1;
- **ENCODING_CDR_ENCAPS**, version 1.2;
- **ENCODING_CDR_ENCAPS** for all future versions of GIOP as they arise.

Vendors are free to support additional encodings.

13.8.2.2 CodecFactory Interface

create_codec

Create a **Codec** of the given encoding.

This operation raises **UnknownEncoding** if this factory cannot create a **Codec** of the given encoding.

Parameter

enc The **Encoding** for which to create a **Codec**.

Return Value

A **Codec** obtained with the given encoding.

13.9 Feature Support and GIOP Versions

The association of service contexts with GIOP versions, (along with some other supported features tied to GIOP minor version), is shown in Table 13-2..

Table 13-2 Feature Support Tied to Minor GIOP Version Number

Feature	Version 1.0	Version 1.1	Version 1.2
TransactionService Service Context	yes	yes	yes
CodeSets Service Context		yes	yes
DCOM Bridging Service Contexts: ChainBypassCheck ChainBypassInfo LogicalThreadId			yes
Object by Value Service Context: SendingContextRunTime			yes
Bi-Directional IIOP Service Context: BI_DIR_IIOP			yes
Asynch Messaging Service Context INVOCATION_POLICIES			optional[$]
Firewall Service Context FORWARDED_IDENTITY			optional[$]
Java Language Throwable Service Context: UnknownExceptionInfo			yes
Realtime CORBA Service Contexts RTCorbaPriority RTCorbaPriorityRange			optional (Realtime CORBA only)
ExceptionDetailMessage Service Context			optional
IOR components in IIOP profile		yes	yes
TAG_ORB_TYPE		yes	yes
TAG_CODE_SETS		yes	yes
TAG_ALTERNATE_IIOP_ADDRESS			yes
TAG_ASSOCIATION_OPTION		yes	yes
TAG_SEC_NAME		yes	yes
TAG_SSL_SEC_TRANS		yes	yes
TAG_GENERIC_SEC_MECH		yes	yes
TAG_*_SEC_MECH		yes	yes
TAG_JAVA_CODEBASE			yes

Table 13-2 Feature Support Tied to Minor GIOP Version Number *(Continued)*

Feature	Version 1.0	Version 1.1	Version 1.2
TAG_FIREWALL_TRANS			optional[$]
TAG_SCCP_CONTACT_INFO			optional[$]
TAG_TRANSACTION_POLICY			optional[$]
TAG_MESSAGE_ROUTERS			optional[$]
TAG_OTS_POLICY			optional[$]
TAG_INV_POLICY			optional[$]
TAG_INET_SEC_TRANS			optional[$]
Extended IDL data types		yes	yes
Bi-Directional GIOP Features			yes
Value types and Abstract Interfaces			yes

Note – [$] All features that have been added after CORBA 2.3 have been marked as optional in GIOP 1.2. These features cannot be compulsory in GIOP 1.2 since there is no way to incorporate them in deployed implementations of 1.2. However, in order to have the additional features of CORBA 2.4 work properly these optional features must be supported by the GIOP 1.2 implementation connecting CORBA 2.4 ORBs.

13.10 Code Set Conversion

13.10.1 Character Processing Terminology

This section introduces a few terms and explains a few concepts to help understand the character processing portions of this document.

13.10.1.1 Character Set

A finite set of different characters used for the representation, organization, or control of data. In this specification, the term "character set" is used without any relationship to code representation or associated encoding. Examples of character sets are the English alphabet, Kanji or sets of ideographic characters, corporate character sets (commonly used in Japan), and the characters needed to write certain European languages.

13.10.1.2 Coded Character Set, or Code Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation or numeric value. In this specification, the term "code set" is used as an abbreviation for the term

"coded character set." Examples include ASCII, ISO 8859-1, JIS X0208 (which includes Roman characters, Japanese hiragana, Greek characters, Japanese kanji, etc.) and Unicode.

13.10.1.3 Code Set Classifications

Some language environments distinguish between byte-oriented and "wide characters." The byte-oriented characters are encoded in one or more 8-bit bytes. A typical single-byte encoding is ASCII as used for western European languages like English. A typical multi-byte encoding which uses from one to three 8-bit bytes for each character is eucJP (Extended UNIX Code - Japan, packed format) as used for Japanese workstations.

Wide characters are a fixed 16 or 32 bits long, and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits is insufficient and a fixed-width encoding is needed. A typical example is Unicode (a "universal" character set defined by the The Unicode Consortium, which uses an encoding scheme identical to ISO 10646 UCS-2, or 2-byte Universal Character Set encoding). An extended encoding scheme for Unicode characters is UTF-16 (UCS Transformation Format, 16-bit representations).

The C language has data types **`char`** for byte-oriented characters and **`wchar_t`** for wide characters. The language definition for C states that the sizes for these characters are implementation-dependent. Some environments do not distinguish between byte-oriented and wide characters (e.g., Ada and Smalltalk). Here again, the size of a character is implementation-dependent. The following table illustrates code set classifications as used in this document.

Table 13-3 Code Set Classification

Orientation	Code Element Encoding	Code Set Examples	C Data Type
byte-oriented	single-byte	ASCII, ISO 8859-1 (Latin-1), EBCDIC, ...	char
	multi-byte	UTF-8, eucJP, Shift-JIS, JIS, Big5, ...	char[]
non-byte-oriented	fixed-length	ISO 10646 UCS-2 (Unicode), ISO 10646 UCS-4, UTF-16, ...	wchar_t

13.10.1.4 Narrow and Wide Characters

Some language environments distinguish between "narrow" and "wide" characters. Typically the narrow characters are considered to be 8-bit long and are used for western European languages like English, while the wide characters are 16-bit or 32-bit long and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits are insufficient. However, as noted above there are common encoding schemes in which Asian characters are encoded using multi-byte code sets and it is incorrect to assume that Asian characters are always encoded as "wide" characters.

Within this specification, the general terms "narrow character" and "wide character" are only used in discussing OMG IDL.

13.10.1.5 Char Data and Wchar Data

The phrase "**char** data" in this specification refers to data whose IDL types have been specified as **char** or **string**. Likewise "**wchar** data" refers to data whose IDL types have been specified as **wchar** or **wstring**.

13.10.1.6 Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy one or more bytes. A byte as used in this specification is synonymous with octet, which occupies 8 bits.

13.10.1.7 Multi-Byte Character Strings

A character string represented in a byte-oriented encoding where each character can occupy one or more bytes is called a multi-byte character string. Typically, wide characters are converted to this form from a (fixed-width) process code set before transmitting the characters outside the process (see below about process code sets). Care must be taken to correctly process the component bytes of a character's multi-byte representation.

13.10.1.8 Non-Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy fixed 16 or 32 bits.

13.10.1.9 Char and Wchar Transmission Code Set (TCS-C and TCS-W)

These two terms refer to code sets that are used for transmission between ORBs after negotiation is completed. As the names imply, the first one is used for **char** data and the second one for **wchar** data. Each TCS can be byte-oriented or non-byte oriented.

13.10.1.10 Process Code Set and File Code Set

Processes generally represent international characters in an internal fixed-width format which allows for efficient representation and manipulation. This internal format is called a "process code set." The process code set is irrelevant outside the process, and hence to the interoperation between CORBA clients and servers through their respective ORBs.

When a process needs to write international character information out to a file, or communicate with another process (possibly over a network), it typically uses a different encoding called a "file code set." In this specification, unless otherwise

indicated, all references to a program's code set refer to the file code set, not the process code set. Even when a client and server are located physically on the same machine, it is possible for them to use different file code sets.

13.10.1.11 Native Code Set

A native code set is the code set which a client or a server uses to communicate with its ORB. There might be separate native code sets for **char** and **wchar** data.

13.10.1.12 Transmission Code Set

A transmission code set is the commonly agreed upon encoding used for character data transfer between a client's ORB and a server's ORB. There are two transmission code sets established per session between a client and its server, one for **char** data (TCS-C) and the other for **wchar** data (TCS-W). Figure 13-6 illustrates these relationships:

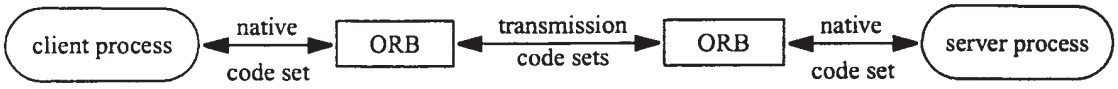

Figure 13-6 Transmission Code Sets

The intent is for TCS-C to be byte-oriented and TCS-W to be non-byte-oriented. However, this specification does allow both types of characters to be transmitted using the same transmission code set. That is, the selection of a transmission code set is orthogonal to the wideness or narrowness of the characters, although a given code set may be better suited for either narrow or wide characters.

13.10.1.13 Conversion Code Set (CCS)

With respect to a particular ORB's native code set, the set of other or target code sets for which an ORB can convert all code points or character encodings between the native code set and that target code set. For each code set in this CCS, the ORB maintains appropriate translation or conversion procedures and advertises the ability to use that code set for transmitted data in addition to the native code set.

13.10.2 Code Set Conversion Framework

13.10.2.1 Requirements

The file code set that an application uses is often determined by the platform on which it runs. In Japan, for example, Japanese EUC is used on Unix systems, while Shift-JIS is used on PCs. Code set conversion is therefore required to enable interoperability across these platforms. This proposal defines a framework for the automatic conversion of code sets in such situations. The requirements of this framework are:

1. Backward compatibility. In previous CORBA specifications, IDL type **char** was limited to ISO 8859-1. The conversion framework should be compatible with existing clients and servers that use ISO 8859-1 as the code set for **char**.

2. Automatic code set conversion. To facilitate development of CORBA clients and servers, the ORB should perform any necessary code set conversions automatically and efficiently. The IDL type **octet** can be used if necessary to prevent conversions.

3. Locale support. An internationalized application determines the code set in use by examining the LOCALE string (usually found in the LANG environment variable), which may be changed dynamically at run time by the user. Example LOCALE strings are fr_FR.ISO8859-1 (French, used in France with the ISO 8859-1 code set) and ja_JP.ujis (Japanese, used in Japan with the EUC code set and X11R5 conventions for LOCALE). The conversion framework should allow applications to use the LOCALE mechanism to indicate supported code sets, and thus select the correct code set from the registry.

4. CMIR and SMIR support. The conversion framework should be flexible enough to allow conversion to be performed either on the client or server side. For example, if a client is running in a memory-constrained environment, then it is desirable for code set converters to reside in the server and for a Server Makes It Right (SMIR) conversion method to be used. On the other hand, if many servers are executed on one server machine, then converters should be placed in each client to reduce the load on the server machine. In this case, the conversion method used is Client Makes It Right (CMIR).

13.10.2.2 Overview of the Conversion Framework

Both the client and server indicate a native code set indirectly by specifying a locale. The exact method for doing this is language-specific, such as the XPG4 C/C++ function `setlocale`. The client and server use their native code set to communicate with their ORB. (Note that these native code sets are in general different from process code sets and hence conversions may be required at the client and server ends.)

The conversion framework is illustrated in Figure 13-7. The server-side ORB stores a server's code set information in a component of the IOR multiple-component profile structure (see Section 13.6.2, "Interoperable Object References: IORs" on page 13-14)[1]. The code sets actually used for transmission are carried in the service context field of an IOP (Inter-ORB Protocol) request header (see Section 13.7, "Service Context" on page 13-28 and Section 13.10.2.5, "GIOP Code Set Service Context" on page 13-43). Recall that there are two code sets (TCS-C and TCS-W) negotiated for each session.

1. Version 1.1 of the IIOP profile body can also be used to specify the server's code set information, as this version introduces an extra field that is a sequence of tagged components.

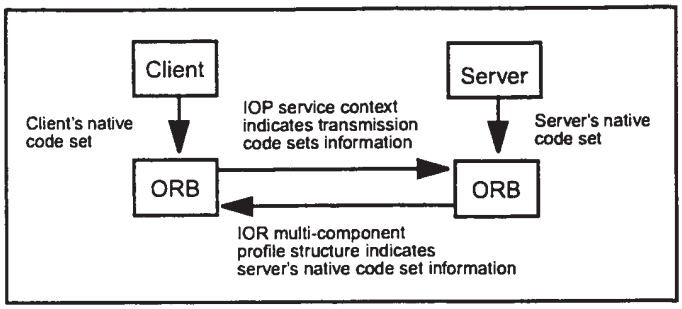

Figure 13-7 Code Set Conversion Framework Overview

If the native code sets used by a client and server are the same, then no conversion is performed. If the native code sets are different and the client-side ORB has an appropriate converter, then the CMIR conversion method is used. In this case, the server's native code set is used as the transmission code set. If the native code sets are different and the client-side ORB does not have an appropriate converter but the server-side ORB does have one, then the SMIR conversion method is used. In this case, the client's native code set is used as the transmission code set.

The conversion framework allows clients and servers to specify a native **char** code set and a native **wchar** code set, which determine the local encodings of IDL types **char** and **wchar**, respectively. The conversion process outlined above is executed independently for the **char** code set and the **wchar** code set. In other words, the algorithm that is used to select a transmission code set is run twice, once for **char** data and once for **wchar** data.

The rationale for selecting two transmission code sets rather than one (which is typically inferred from the locale of a process) is to allow efficient data transmission without any conversions when the client and server have identical representations for **char** and/or **wchar** data. For example, when a Windows NT client talks to a Windows NT server and they both use Unicode for wide character data, it becomes possible to transmit wide character data from one to the other without any conversions. Of course, this becomes possible only for those wide character representations that are well-defined, not for any proprietary ones. If a single transmission code set was mandated, it might require unnecessary conversions. (For example, choosing Unicode as the transmission code set would force conversion of all byte-oriented character data to Unicode.)

13.10.2.3 ORB Databases and Code Set Converters

The conversion framework requires an ORB to be able to determine the native code set for a locale and to convert between code sets as necessary. While the details of exactly how these tasks are accomplished are implementation-dependent, the following databases and code set converters might be used:

- Locale database. This database defines a native code set for a process. This code set could be byte-oriented or non-byte-oriented and could be changed programmatically while the process is running. However, for a given session between a client and a server, it is fixed once the code set information is negotiated at the session's setup time.
- Environment variables or configuration files. Since the locale database can only indicate one code set while the ORB needs to know two code sets, one for **char** data and one for **wchar** data, an implementation can use environment variables or configuration files to contain this information on native code sets.
- Converter database. This database defines, for each code set, the code sets to which it can be converted. From this database, a set of "conversion code sets" (CCS) can be determined for a client and server. For example, if a server's native code set is eucJP, and if the server-side ORB has eucJP-to-JIS and eucJP-to-SJIS bilateral converters, then the server's conversion code sets are JIS and SJIS.
- Code set converters. The ORB has converters which are registered in the converter database.

13.10.2.4 CodeSet Component of IOR Multi-Component Profile

The code set component of the IOR multi-component profile structure contains:

- server's native **char** code set and conversion code sets, and
- server's native **wchar** code set and conversion code sets.

Both **char** and **wchar** conversion code sets are listed in order of preference. The code set component is identified by the following tag:

```
const IOP::ComponentID TAG_CODE_SETS = 1;
```

This tag has been assigned by OMG (See Section 13.6.6, "Standard IOR Components" on page 13-19.). The following IDL structure defines the representation of code set information within the component:

```
module CONV_FRAME {                          // IDL
    typedef unsigned long CodeSetId;
    struct CodeSetComponent {
        CodeSetId               native_code_set;
        sequence<CodeSetId>     conversion_code_sets;
    };
    struct CodeSetComponentInfo {

        CodeSetComponent        ForCharData;
        CodeSetComponent        ForWcharData;
    };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See Section 13.10.5.1, "Character and Code Set Registry" on page 13-49 for further information). Data within the code set component is represented as a structure

of type **CodeSetComponentInfo**, and is encoded as a CDR encapsulation. In other words, the **char** code set information comes first, then the **wchar** information, represented as structures of type **CodeSetComponent**.

A null value should be used in the **native_code_set** field if the server desires to indicate no native code set (possibly with the identification of suitable conversion code sets).

If the code set component is not present in a multi-component profile structure, then the default **char** code set is ISO 8859-1 for backward compatibility. However, there is no default **wchar** code set. If a server supports interfaces that use wide character data but does not specify the **wchar** code sets that it supports, client-side ORBs will raise exception INV_OBJREF, with standard minor code 1.

If a client application invokes an operation which results in an attempt by the client ORB to marshall wchar or wstring data for an in parameter (or to unmarshal wchar or wstring data for an in/out parameter, out parameter or the return value), and the associated Object Reference does not include a codeset component, then the client ORB shall raise the INV_OBJREF standard system exception with standard minor code 2 as a response to the operation invocation.

13.10.2.5 GIOP Code Set Service Context

The code set GIOP service context contains:

- **char** transmission code set, and
- **wchar** transmission code set

in the form of a code set service. This service is identified by:

```
const IOP::ServiceID  CodeSets = 1;
```

The following IDL structure defines the representation of code set service information:

```
module CONV_FRAME {                          // IDL
    typedef unsigned long CodeSetId;
    struct CodeSetContext {
        CodeSetId           char_data;
        CodeSetId           wchar_data;
    };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See Section 13.10.5.1, "Character and Code Set Registry" on page 13-49 for further information).

Note – A server's **char** and **wchar** Code set components are usually different, but under some special circumstances they can be the same. That is, one could use the same code set for both **char** data and **wchar** data. Likewise the **CodesetIds** in the service context don't have to be different.

13.10.2.6 Code Set Negotiation

The client-side ORB determines a server's native and conversion code sets from the code set component in an IOR multi-component profile structure, and it determines a client's native and conversion code sets from the locale setting (and/or environment variables/configuration files) and the converters that are available on the client. From this information, the client-side ORB chooses **char** and **wchar** transmission code sets (TCS-C and TCS-W). For both requests and replies, the **char** TCS-C determines the encoding of **char** and **string** data, and the **wchar** TCS-W determines the encoding of **wchar** and **wstring** data.

Code set negotiation is not performed on a per-request basis, but only when a client initially connects to a server. All text data communicated on a connection are encoded as defined by the TCSs selected when the connection is established.

Figure 13-8 illustrates, there are two channels for character data flowing between the client and the server. The first, TCS-C, is used for **char** data and the second, TCS-W, is used for **wchar** data. Also note that two native code sets, one for each type of data, could be used by the client and server to talk to their respective ORBs (as noted earlier, the selection of the particular native code set used at any particular point is done via `setlocale` or some other implementation-dependent method).

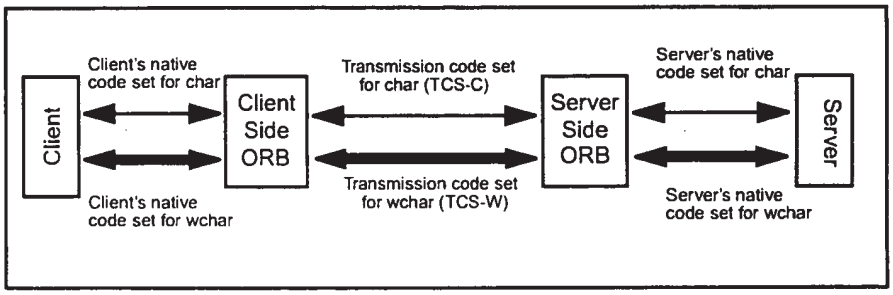

Figure 13-8 Transmission Code Set Use

Let us look at an example. Assume that the code set information for a client and server is as shown in the table below. (Note that this example concerns only **char** code sets and is applicable only for data described as **char**s in the IDL.)

	Client	**Server**
Native code set:	SJIS	eucJP
Conversion code sets:	eucJP, JIS	SJIS, JIS

The client-side ORB first compares the native code sets of the client and server. If they are identical, then the transmission and native code sets are the same and no conversion is required. In this example, they are different, so code set conversion is necessary. Next, the client-side ORB checks to see if the server's native code set, eucJP, is one of the conversion code sets supported by the client. It is, so eucJP is selected as the

transmission code set, with the client (i.e., its ORB) performing conversion to and from its native code set, SJIS, to eucJP. Note that the client may first have to convert all its data described as **char**s (and possibly **wchar_t**s) from process codes to SJIS first.

Now let us look at the general algorithm for determining a transmission code set and where conversions are performed. First, we introduce the following abbreviations:

- CNCS - Client Native Code Set;
- CCCS - Client Conversion Code Sets;
- SNCS - Server Native Code Set;
- SCCS - Server Conversion Code Sets; and
- TCS - Transmission Code Set.

The algorithm is as follows:

```
if (CNCS==SNCS)
        TCS = CNCS;                 // no conversion required
else {
        if (elementOf(SNCS,CCCS))
            TCS = SNCS;   // client converts to server's native code set
        else if (elementOf(CNCS,SCCS))
            TCS = CNCS;   // server converts from client's native code set
        else if (intersection(CCCS,SCCS) != emptySet) {
            TCS = oneOf(intersection(CCCS,SCCS));
            // client chooses TCS, from intersection(CCCS,SCCS), that is
            // most preferable to server;
            // client converts from CNCS to TCS and server
            // from TCS to SNCS
        else if (compatible(CNCS,SNCS))
            TCS = fallbackCS;       // fallbacks are UTF-8 (for char data) and
                                    // UTF-16 (for wchar data)
        else
            raise CODESET_INCOMPATIBLE exception;
}
```

The algorithm first checks to see if the client and server native code sets are the same. If they are, then the native code set is used for transmission and no conversion is required. If the native code sets are not the same, then the conversion code sets are examined to see if

1. the client can convert from its native code set to the server's native code set,
2. the server can convert from the client's native code set to its native code set, or
3. transmission through an intermediate conversion code set is possible.

If the third option is selected and there is more than one possible intermediate conversion code set (i.e., the intersection of CCCS and SCCS contains more than one code set), then the one most preferable to the server is selected.[2]

If none of these conversions is possible, then the fallback code set (UTF-8 for **char** data and UTF-16 for **wchar** data— see below) is used. However, before selecting the fallback code set, a compatibility test is performed. This test looks at the character sets encoded by the client and server native code sets. If they are different (e.g., Korean and French), then meaningful communication between the client and server is not possible and a CODESET_INCOMPATIBLE exception is raised. This test is similar to the DCE compatibility test and is intended to catch those cases where conversion from the client native code set to the fallback, and the fallback to the server native code set would result in massive data loss. (See Section 13.10.5, "Relevant OSFM Registry Interfaces" on page 13-49 for the relevant OSF registry interfaces that could be used for determining compatibility.)

A DATA_CONVERSION exception is raised when a client or server attempts to transmit a character that does not map into the negotiated transmission code set. For example, not all characters in Taiwan Chinese map into Unicode. When an attempt is made to transmit one of these characters via Unicode, an ORB is required to raise a DATA_CONVERSION exception, with standard minor code 1.

In summary, the fallback code set is UTF-8 for **char** data (identified in the Registry as 0x05010001, "X/Open UTF-8; UCS Transformation Format 8 (UTF-8)"), and UTF-16 for **wchar** data (identified in the Registry as 0x00010109, "ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit form"). As mentioned above the fallback code set is meaningful only when the client and server character sets are compatible, and the fallback code set is distinguished from a default code set used for backward compatibility.

If a server's native **char** code set is not specified in the IOR multi-component profile, then it is considered to be ISO 8859-1 for backward compatibility. However, a server that supports interfaces that use wide character data is required to specify its native **wchar** code set; if one is not specified, then the client-side ORB raises exception INV_OBJREF, with standard minor code set to 1.

Similarly, if no **char** transmission code set is specified in the code set service context, then the **char** transmission code set is considered to be ISO 8859-1 for backward compatibility. If a client transmits wide character data and does not specify its **wchar** transmission code set in the service context, then the server-side ORB raises exception BAD_PARAM, with standard minor code set to 23.

To guarantee "out-of-the-box" interoperability, clients and servers must be able to convert between their native **char** code set and UTF-8 and their native **wchar** code set (if specified) and Unicode. Note that this does not require that all server native code sets be mappable to Unicode, but only those that are exported as native in the IOR. The server may have other native code sets that aren't mappable to Unicode and those can

---

2. Recall that server conversion code sets are listed in order of preference.

be exported as SCCSs (but not SNCSs). This is done to guarantee out-of-the-box interoperability and to reduce the number of code set converters that a CORBA-compliant ORB must provide.

ORB implementations are strongly encouraged to use widely-used code sets for each regional market. For example, in the Japanese marketplace, all ORB implementations should support Japanese EUC, JIS and Shift JIS to be compatible with existing business practices.

13.10.3 Mapping to Generic Character Environments

Certain language environments do not distinguish between byte-oriented and wide characters. In such environments both **char** and **wchar** are mapped to the same "generic" character representation of the language. **String** and **wstring** are likewise mapped to generic strings in such environments. Examples of language environments that provide generic character support are Smalltalk and Ada.

Even while using languages that do distinguish between wide and byte-oriented characters (e.g., C and C++), it is possible to mimic some generic behavior by the use of suitable macros and support libraries. For example, developers of Windows NT and Windows 95 applications can write portable code between NT (which uses Unicode strings) and Windows 95 (which uses byte-oriented character strings) by using a set of macros for declaring and manipulating characters and character strings. Appendix A in this chapter shows how to map wide and byte-oriented characters to these generic macros.

Another way to achieve generic manipulation of characters and strings is by treating them as abstract data types (ADTs). For example, if strings were treated as abstract data types and the programmers are required to create, destroy, and manipulate strings only through the operations in the ADT interface, then it becomes possible to write code that is representation-independent. This approach has an advantage over the macro-based approach in that it provides portability between byte-oriented and wide character environments even without recompilation (at runtime the string function calls are bound to the appropriate byte-oriented/wide library). Another way of looking at it is that the macro-based genericity gives compile-time flexibility, while ADT-based genericity gives runtime flexibility.

Yet another way to achieve generic manipulation of character data is through the ANSI C++ Strings library defined as a template that can be parameterized by `char`, `wchar_t`, or other integer types.

Given that there can be several ways of treating characters and character strings in a generic way, this standard cannot, and therefore does not, specify the mapping of **char**, **wchar**, **string**, and **wstring** to all of them. It only specifies the following normative requirements which are applicable to generic character environments:

- **wchar** must be mapped to the generic character type in a generic character environment.
- **wstring** must be mapped to a string of such generic characters in a generic character environment.

- The language binding files (i.e., stubs) generated for these generic environments must ensure that the generic type representation is converted to the appropriate code sets (i.e., CNCS on the client side and SNCS on the server side) before character data is given to the ORB runtime for transmission.

13.10.3.1 Describing Generic Interfaces

To describe generic interfaces in IDL we recommend using **wchar** and **wstring**. These can be mapped to generic character types in environments where they do exist and to wide characters where they do not. Either way interoperation between generic and non-generic character type environments is achieved because of the code set conversion framework.

13.10.3.2 Interoperation

Let us consider an example to see how a generic environment can interoperate with a non-generic environment. Let us say there is an IDL interface with both **char** and **wchar** parameters on the operations, and let us say the client of the interface is in a generic environment while the server is in a non-generic environment (for example the client is written in Smalltalk and the server is written in C++).

Assume that the server's (byte-oriented) native **char** code set (SNCS) is eucJP and the client's native **char** code set (CNCS) is SJIS. Further assume that the code set negotiation led to the decision to use eucJP as the **char** TCS-C and Unicode as the **wchar** TCS-W.

As per the above normative requirements for mapping to a generic environment, the client's Smalltalk stubs are responsible for converting all **char** data (however they are represented inside Smalltalk) to SJIS and all **wchar** data to the client's **wchar** code set before passing the data to the client-side ORB. Note that this conversion could be an identity mapping if the internal representation of narrow and wide characters is the same as that of the native code set(s). The client-side ORB now converts all **char** data from SJIS to eucJP and all **wchar** data from the client's **wchar** code set to Unicode, and then transmits to the server side.

The server side ORB and stubs convert the eucJP data and Unicode data into C++'s internal representation for **char**s and **wchar**s as dictated by the IDL operation signatures. Notice that when the data arrives at the server side it does not look any different from data arriving from a non-generic environment (e.g., that is just like the server itself). In other words, the mappings to generic character environments do not affect the code set conversion framework.

13.10.4 Example of Generic Environment Mapping

This section shows how **char**, **wchar**, **string**, and **wchar** can be mapped to the generic C/C++ macros of the Windows environment. This is merely to illustrate one possibility. This section is not normative and is applicable only in generic environments. See Section 13.10.3, "Mapping to Generic Character Environments" on page 13-47.

13.10.4.1 Generic Mappings

Char and **string** are mapped to C/C++ `char` and `char*` as per the standard C/C++ mappings. **wchar** is mapped to the `TCHAR` macro which expands to either `char` or `wchar_t` depending on whether `_UNICODE` is defined. **wstring** is mapped to pointers to `TCHAR` as well as to the string class `CORBA::Wstring_var`. Literal strings in IDL are mapped to the `_TEXT` macro as in `_TEXT(<literal>)`.

13.10.4.2 Interoperation and Generic Mappings

We now illustrate how the interoperation works with the above generic mapping. Consider an IDL interface operation with a **wstring** parameter, a client for the operation which is compiled and run on a Windows 95 machine, and a server for the operation which is compiled and run on a Windows NT machine. Assume that the locale (and/or the environment variables for CNCS for **wchar** representation) on the Windows 95 client indicates the client's native code set to be SJIS, and that the corresponding server's native code set is Unicode. The code set negotiation in this case will probably choose Unicode as the TCS-W.

Both the client and server sides will be compiled with `_UNICODE` defined. The IDL type **wstring** will be represented as a string of `wchar_t` on the client. However, since the client's locale or environment indicates that the CNCS for wide characters is SJIS, the client side ORB will get the **wstring** parameter encoded as a SJIS multi-byte string (since that is the client's native code set), which it will then convert to Unicode before transmitting to the server. On the server side the ORB has no conversions to do since the TCS-W matches the server's native code set for wide characters.

We therefore notice that the code set conversion framework handles the necessary translations between byte-oriented and wide forms.

13.10.5 Relevant OSFM Registry Interfaces

13.10.5.1 Character and Code Set Registry

The OSF character and code set registry is defined in *OSF Character and Code Set Registry* (see References in the Preface) and current registry contents may be obtained directly from the Open Software Foundation (obtain via anonymous ftp to ftp.opengroup.org:/pub/code_set_registry). This registry contains two parts: character sets and code sets. For each listed code set, the set of character sets encoded by this code set is shown.

Each 32-bit code set value consists of a high-order 16-bit organization number and a 16-bit identification of the code set within that organization. As the numbering of organizations starts with 0x0001, a code set null value (0x00000000) may be used to indicate an unknown code set.

When associating character sets and code sets, OSF uses the concept of "fuzzy equality," meaning that a code set is shown as encoding a particular character set if the code set can encode "most" of the characters.

"Compatibility" is determined with respect to two code sets by examining their entries in the registry, paying special attention to the character sets encoded by each code set. For each of the two code sets, an attempt is made to see if there is at least one (fuzzy-defined) character set in common, and if such a character set is found, then the assumption is made that these code sets are "compatible." Obviously, applications which exploit parts of a character set not properly encoded in this scheme will suffer information loss when communicating with another application in this "fuzzy" scheme.

The ORB is responsible for accessing the OSF registry and determining "compatibility" based on the information returned.

OSF members and other organizations can request additions to both the character set and code set registries by email to *cs-registry@opengroup.org*; in particular, one range of the code set registry (`0xf5000000` through `0xffffffff`) is reserved for organizations to use in identifying sets which are not registered with the OSF (although such use would not facilitate interoperability without registration).

13.10.5.2 Access Routines

The following routines are for accessing the OSF character and code set registry. These routines map a code set string name to code set id and vice versa. They also help in determining character set compatibility. These routine interfaces, their semantics and their actual implementation are not normative (i.e., ORB vendors do not have to bundle the OSF registry implementation with their products for compliance).

The following routines are adopted from *RPC Runtime Support For I18N Characters - Functional Specification* (see References in the Preface).

dce_cs_loc_to_rgy

Maps a local system-specific string name for a code set to a numeric code set value specified in the code set registry.

Synopsis

```
void dce_cs_loc_to_rgy(
    idl_char *local_code_set_name,
    unsigned32 *rgy_code_set_value,
    unsigned16 *rgy_char_sets_number,
    unsigned16 **rgy_char_sets_value,
    error_status_t *status);
```

Parameters

Input

local_code_set_name - A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes plus a terminating NULL character.

Output

rgy_code_set_value 0 - The registered integer value that uniquely identifies the code set specified by local_code_set_name.

rgy_char_sets_number - The number of character sets that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter.

rgy_char_sets_value - A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter. The routine dynamically allocates this value.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- dce_cs_c_ok – Code set registry access operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – No code set value was not found in the registry which corresponds to the code set name specified.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the name specified.

Description

The dce_cs_loc_to_rgy() routine maps operating system-specific names for character/code set encodings to their unique identifiers in the code set registry.

The dce_cs_loc_to_rgy() routine takes as input a string that holds the host-specific "local name" of a code set and returns the corresponding integer value that uniquely identifies that code set, as registered in the host's code set registry. If the integer value does not exist in the registry, the routine returns the status dce_cs_c_unknown.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a code set value from the code set registry can specify NULL for these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the array after it is used, since the array is dynamically allocated.

dce_cs_rgy_to_loc

Maps a numeric code set value contained in the code set registry to the local system-specific name for a code set.

Synopsis

```
void dce_cs_rgy_to_loc(
        unsigned32 *rgy_code_set_value,
        idl_char **local_code_set_name,
        unsigned16 *rgy_char_sets_number,
        unsigned16 **rgy_char_sets_value,
        error_status_t *status);
```

Parameters

Input

rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set.

Output

local_code_set_name - A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes and a terminating NULL character.

rgy_char_sets_number - The number of character sets that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value.

rgy_char_sets_value - A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value. The routine dynamically allocates this value.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- dce_cs_c_ok – Code set registry access operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – The requested code set value was not found in the code set registry.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the specific code set registry ID value. This implies that the code set is not supported in the local system environment.

Description

The dce_cs_rgy_to_loc() routine maps a unique identifier for a code set in the code set registry to the operating system-specific string name for the code set, if it exists in the code set registry.

The dce_cs_rgy_to_loc() routine takes as input a registered integer value of a code set and returns a string that holds the operating system-specific, or local name, of the code set.

If the code set identifier does not exist in the registry, the routine returns the status dce_cs_c_unknown and returns an undefined string.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a local code set name from the code set registry can specify NULL for

these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the rgy_char_sets_value array after it is used.

rpc_cs_char_set_compat_check

Evaluates character set compatibility between a client and a server.

Synopsis

```
void rpc_cs_char_set_compat_check(
        unsigned32 client_rgy_code_set_value,
        unsigned32 server_rgy_code_set_value,
        error_status_t *status);
```

Parameters

Input

client_rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set that the client is using as its local code set.

server_rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set that the server is using as its local code set.

Output

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- rpc_s_ok – Successful status.
- rpc_s_ss_no_compat_charsets – No compatible code set found. The client and server do not have a common encoding that both could recognize and convert.
- The routine can also return status codes from the dce_cs_rgy_to_loc() routine.

Description

The rpc_cs_char_set_compat_check() routine provides a method for determining character set compatibility between a client and a server; if the server's character set is incompatible with that of the client, then connecting to that server is most likely not acceptable, since massive data loss would result from such a connection.

The routine takes the registered integer values that represent the code sets that the client and server are currently using and calls the code set registry to obtain the registered values that represent the character set(s) that the specified code sets support. If both client and server support just one character set, the routine compares client and server registered character set values to determine whether or not the sets are compatible. If they are not, the routine returns the status message rpc_s_ss_no_compat_charsets.

If the client and server support multiple character sets, the routine determines whether at least two of the sets are compatible. If two or more sets match, the routine considers the character sets compatible, and returns a success status code to the caller.

rpc_rgy_get_max_bytes

Gets the maximum number of bytes that a code set uses to encode one character from the code set registry on a host

Synopsis

```
void rpc_rgy_get_max_bytes(
    unsigned32 rgy_code_set_value,
    unsigned16 *rgy_max_bytes,
    error_status_t *status);
```

Parameters

Input

rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set.

Output

rgy_max_bytes - The registered decimal value that indicates the number of bytes this code set uses to encode one character.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- rpc_s_ok – Operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – No code set value was not found in the registry which corresponds to the code set value specified.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the value specified.

Description

The rpc_rgy_get_max_bytes() routine reads the code set registry on the local host. It takes the specified registered code set value, uses it as an index into the registry, and returns the decimal value that indicates the number of bytes that the code set uses to encode one character.

This information can be used for buffer sizing as part of the procedure to determine whether additional storage needs to be allocated for conversion between local and network code sets.

The DCE ESIOP 16

This chapter specifies an Environment-Specific Inter-ORB Protocol (ESIOP) for the OSF DCE environment, the DCE Common Inter-ORB Protocol (DCE-CIOP).

Contents

This chapter contains the following sections.

16.1 Goals of the DCE Common Inter-ORB Protocol

DCE CIOP was designed to meet the following goals:

- Support multi-vendor, mission-critical, enterprise-wide, ORB-based applications.
- Leverage services provided by DCE wherever appropriate.
- Allow efficient and straightforward implementation using public DCE APIs.
- Preserve ORB implementation freedom.

DCE CIOP achieves these goals by using DCE-RPC to provide message transport, while leaving the ORB responsible for message formatting, data marshaling, and operation dispatch.

16.2 DCE Common Inter-ORB Protocol Overview

The DCE Common Inter-ORB Protocol uses the wire format and RPC packet formats defined by DCE-RPC to enable independently implemented ORBs to communicate. It defines the message formats that are exchanged using DCE-RPC, and specifies how information in object references is used to establish communication between client and server processes.

The full OMG IDL for the DCE ESIOP specification is shown in Section 16.7, "OMG IDL for the DCE CIOP Module," on page 16-25. Fragments are used throughout this chapter as necessary.

16.2.1 DCE-CIOP RPC

DCE-CIOP requires an RPC, which is interoperable with the DCE connection-oriented and/or connectionless protocols as specified in the X/Open *CAE Specification C309* and the OSF *AES/Distributed Computing RPC Volume*. Some of the features of the DCE-RPC are as follows:

- Defines connection-oriented and connectionless protocols for establishing the communication between a client and server.
- Supports multiple underlying transport protocols including TCP/IP.
- Supports multiple outstanding requests to multiple CORBA objects over the same connection.
- Supports fragmentation of messages. This provides for buffer management by ORBs of CORBA requests, which contain a large amount of marshaled data.

All interactions between ORBs take the form of remote procedure calls on one of two well-known DCE-RPC interfaces. Two DCE operations are provided in each interface:

- *invoke* - for invoking CORBA operation requests, and
- *locate* - for locating server processes.

Each DCE operation is a synchronous remote procedure call[1,2], consisting of a request message being transmitted from the client to the server, followed by a response message being transmitted from the server to the client.

---

1. DCE *maybe* operation semantics cannot be used for CORBA *oneway* operations because they are idempotent as opposed to at-most-once.
2. The deferred synchronous DII API can be implemented on top of synchronous RPCs by using threads.

Using one of the DCE-RPC interfaces, the messages are transmitted as pipes of uninterpreted bytes. By transmitting messages via DCE pipes, the following characteristics are achieved:

- Large amounts of data can be transmitted efficiently.
- Buffering of complete messages is not required.
- Marshaling and demarshaling can take place concurrently with message transmission.
- Encoding of messages and marshaling of data is completely under the control of the ORB.
- DCE client and server stubs can be used to implement DCE-CIOP.

Using the other DCE-RPC interface, the messages are transmitted as conformant arrays of uninterpreted bytes. This interface does not offer all the advantages of the pipe-based interface, but is provided to enable interoperability with ORBs using DCE-RPC implementations that do not adequately support pipes.

16.2.2 DCE-CIOP Data Representation

DCE-CIOP messages represent OMG IDL types by using the CDR transfer syntax, which is defined in Section 15.2.1, "Common Data Representation (CDR)," on page 15-3. DCE-CIOP message headers and bodies are specified as OMG IDL types. These are encoded using CDR, and the resulting messages are passed between client and server processes via DCE-RPC pipes or conformant arrays.

NDR is the transfer syntax used by DCE-RPC for operations defined in DCE IDL. CDR, used to represent messages defined in OMG IDL on top of DCE RPCs, represents the OMG IDL primitive types identically to the NDR representation of corresponding DCE IDL primitive types.

The corresponding OMG IDL and DCE IDL primitive types are shown in Table 16-1.

Table 16-1 Relationship between CDR and NDR primitive data types

OMG IDL type	**DCE IDL type with NDR representation equivalent to CDR representation of OMG IDL type**
char	byte
wchar	byte, unsigned short, or unsigned long, depending on transmission code set
octet	byte
short	short
unsigned short	unsigned short
long	long
unsigned long	unsigned long

Table 16-1 Relationship between CDR and NDR primitive data types

OMG IDL type	DCE IDL type with NDR representation equivalent to CDR representation of OMG IDL type
long long	hyper
unsigned long long	unsigned hyper
float	float[1]
double	double[2]
long double	long double[3]
boolean	byte[4]

1. Restricted to IEEE format.
2. Restricted to IEEE format.
3. Restricted to IEEE format.
4. Values restricted to 0 and 1.

The CDR representation of OMG IDL constructed types and pseudo-object types does not correspond to the NDR representation of types describable in DCE IDL.

A wide string is encoded as a unidimensional conformant array of octets in DCE 1.1 NDR. This consists of an unsigned long of four octets, specifying the number of octets in the array, followed by that number of octets, with no null terminator.

The NDR representation of OMG IDL fixed-point type, **fixed**, will be proposed as an addition to the set of DCE IDL types.

As new data types are added to OMG IDL, NDR can be used as a model for their CDR representations.

16.2.3 DCE-CIOP Messages

The following request and response messages are exchanged between ORB clients and servers via the `invoke` and `locate` RPCs:

- *Invoke Request* identifies the target object and the operation and contains the principal, the operation context, a **ServiceContext**, and the **in** and **inout** parameter values.
- *Invoke Response* indicates whether the operation succeeded, failed, or needs to be reinvoked at another location, and returns a **ServiceContext**. If the operation succeeded, the result and the **out** and **inout** parameter values are returned. If it failed, an exception is returned. If the object is at another location, new RPC binding information is returned.

- *Locate Request* identifies the target object and the operation.
- *Locate Response* indicates whether the location is in the current process, is elsewhere, or is unknown. If the object is at another location, new RPC binding information is returned.

All message formats begin with a field that indicates the byte order used in the CDR encoding of the remainder of the message. The CDR byte order of a message is required to match the NDR byte order used by DCE-RPC to transmit the message.

16.2.4 Interoperable Object Reference (IOR)

For DCE-CIOP to be used to invoke operations on an object, the information necessary to reference an object via DCE-CIOP must be included in an IOR. This information can coexist with the information needed for other protocols such as IIOP. DCE-CIOP information is stored in an IOR as a set of components in a profile identified by either **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS**. Components are defined for the following purposes:

- To identify a server process via a DCE string binding, which can be either fully or partially bound. This process may be a server process implementing the object, or it may be an agent capable of locating the object implementation.
- To identify a server process via a name that can be resolved using a DCE nameservice. Again, this process may implement the object or may be an agent capable of locating it.
- In the **TAG_MULTIPLE_COMPONENTS** profile, to identify the target object when request messages are sent to the server. In the **TAG_INTENET_IOP** profile, the **object_key** profile member is used instead.
- To enable a DCE-CIOP client to recognize objects that share an endpoint.
- To indicate whether a DCE-CIOP client should send a locate message or an invoke message.
- To indicate if the pipe-based DCE-RPC interface is not available.

The IOR is created by the server ORB to provide the information necessary to reference the CORBA object.

16.3 DCE-CIOP Message Transport

DCE-CIOP defines two DCE-RPC interfaces for the transport of messages between client ORBs and server ORBs[3]. One interface uses pipes to convey the messages, while the other uses conformant arrays.

The pipe-based interface is the preferred interface, since it allows messages to be transmitted without precomputing the message length. But not all DCE-RPC implementations adequately support pipes, so this interface is optional. All client and server ORBs implementing DCE-CIOP must support the array-based interface[4].

While server ORBs may provide both interfaces or just the array-based interface, it is up to the client ORB to decide which to use for an invocation. If a client ORB tries to use the pipe-based interface and receives an **rpc_s_unknown_if** error, it should fall back to the array-based interface.

16.3.1 Pipe-based Interface

The **dce_ciop_pipe** interface is defined by the DCE IDL specification shown below:

```
[/* DCE IDL */
uuid(d7d99f66-97ee-11cf-b1a0-0800090b5d3e),/* 2nd revision
*/
version(1.0)
]
interface dce_ciop_pipe
{
typedef pipe byte message_type;
      void invoke  ( [in]  handle_t binding_handle,
                     [in]  message_type *request_message,
                     [out] message_type *response_message);
      void locate  ( [in]  handle_t binding_handle,
                     [in]  message_type *request_message,
                     [out] message_type *response_message);
}
```

ORBs can implement the **dce_ciop_pipe** interface by using DCE stubs generated from this IDL specification, or by using lower-level APIs provided by a particular DCE-RPC implementation.

The **dce_ciop_pipe** interface is identified by the UUID and version number shown. To provide maximal performance, all server ORBs and location agents implementing DCE-CIOP should listen for and handle requests made to this interface. To maximize the chances of interoperating with any DCE-CIOP client, servers should listen for requests arriving via all available DCE protocol sequences.

Client ORBs can invoke OMG IDL operations over DCE-CIOP by performing DCE RPCs on the **dce_ciop_pipe** interface. The **dce_ciop_pipe** interface is made up of two DCE-RPC operations, **invoke** and **locate**. The first parameter of each of these RPCs is a DCE binding handle, which identifies the server process on which to

---

3. Previous DCE-CIOP revisions used different DCE RPC interface UUIDs and had incompatible message formats. These previous revisions are deprecated, but implementations can continue to support them in conjunction with the current interface UUIDs and message formats.

4. A future DCE-CIOP revision may eliminate the array-based interface and require support of the pipe-based interface.

perform the RPC. See "DCE-CIOP String Binding Component" on page 16-17, "DCE-CIOP Binding Name Component" on page 16-18, and "DCE-CIOP Object Location" on page 16-21 for discussion of how these binding handles are obtained. The remaining parameters of the **dce_ciop_pipe** RPCs are pipes of uninterpreted bytes. These pipes are used to convey messages encoded using CDR. The **request_message** input parameters send a request message from the client to the server, while the **response_message** output parameters return a response message from the server to the client.

Figure 16-1 illustrates the layering of DCE-CIOP messages on the DCE-RPC protocol as NDR pipes:

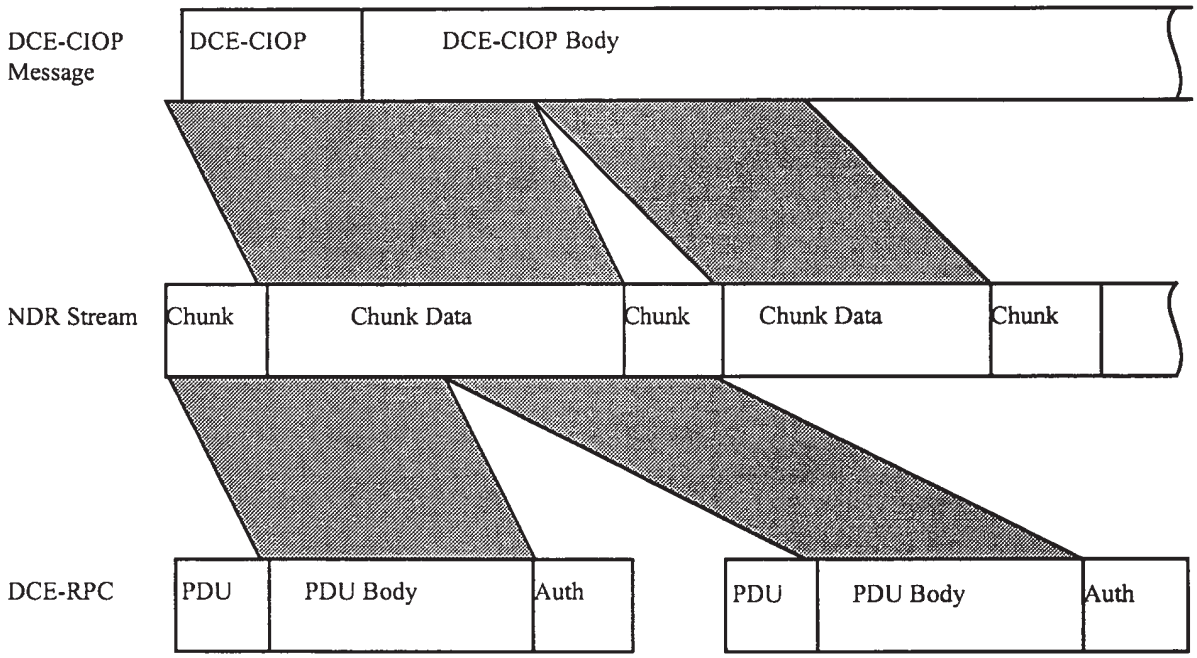

Figure 16-1 Pipe-based Interface Protocol Layering

The DCE-RPC protocol data unit (PDU) bodies, after any appropriate authentication is performed, are concatenated by the DCE-RPC run-time to form an NDR stream. This stream is then interpreted as the NDR representation of a DCE IDL pipe.

A pipe is made up of chunks, where each chunk consists of a chunk length and chunk data. The chunk length is an unsigned long indicating the number of pipe elements that make up the chunk data. The pipe elements are DCE IDL bytes, which are uninterpreted by NDR. A pipe is terminated by a chunk length of zero. The pipe chunks are concatenated to form a DCE-CIOP message.

16.3.1.1 Invoke

The **invoke** RPC is used by a DCE-CIOP client process to attempt to invoke a CORBA operation in the server process identified by the **binding_handle** parameter. The **request_message** pipe transmits a DCE-CIOP invoke request message, encoded using CDR, from the client to the server. See Section 16.4.1, "DCE_CIOP Invoke Request Message," on page 16-11 for a description of its format. The **response_message** pipe transmits a DCE-CIOP invoke response message, also encoded using CDR, from the server to the client. See Section 16.4.2, "DCE-CIOP Invoke Response Message," on page 16-12 for a description of the response format.

16.3.1.2 Locate

The **locate RPC** is used by a DCE-CIOP client process to query the server process identified by the **binding_handle** parameter for the location of the server process where requests should be sent. The **request_message** and **response_message** parameters are used similarly to the parameters of the **invoke RPC**. See Section 16.4.3, "DCE-CIOP Locate Request Message," on page 16-14 and Section 16.4.4, "DCE-CIOP Locate Response Message," on page 16-15 for descriptions of their formats. Use of the **locate RPC** is described in detail in Section 16.6, "DCE-CIOP Object Location," on page 16-21.

16.3.2 Array-based Interface

The **dce_ciop_array** interface is defined by the DCE IDL specification shown below:

```
[ /* DCE IDL */
uuid(09f9ffb8-97ef-11cf-9c96-0800090b5d3e),/* 2nd revision
*/
version(1.0)
]
interface dce_ciop_array
{
    typedef struct {
        unsigned long length;
        [size_is(length),ptr] byte *data;
    } message_type;

    void invoke     ( [in] handle_t binding_handle,
                      [in] message_type *request_message,
                      [out] message_type *response_message);

    void locate     ( [in] handle_t binding_handle,
                      [in] message_type *request_message,
                      [out] message_type *response_message);
}
```

ORBs can implement the **dce_ciop_array** interface, identified by the UUID and version number shown, by using DCE stubs generated from this IDL specification, or by using lower-level APIs provided by a particular DCE-RPC implementation.

All server ORBs and location agents implementing DCE-CIOP must listen for and handle requests made to the **dce_ciop_array** interface, and to maximize interoperability, should listen for requests arriving via all available DCE protocol sequences.

Client ORBs can invoke OMG IDL operations over DCE-CIOP by performing **locate** and **invoke** RPCs on the **dce_ciop_array** interface.

As with the **dce_ciop_pipe** interface, the first parameter of each **dce_ciop_array** RPC is a DCE binding handle that identifies the server process on which to perform the RPC. The remaining parameters are structures containing CDR-encoded messages. The **request_message** input parameters send a request message from the client to the server, while the **response_message** output parameters return a response message from the server to the client.

The **message_type** structure used to convey messages is made up of a **length** member and a **data** member:

- *length* - This member indicates the number of bytes in the message.
- *data* - This member is a full pointer to the first byte of the conformant array containing the message.

The layering of DCE-CIOP messages on DCE-RPC using NDR arrays is illustrated in Figure 16-2:

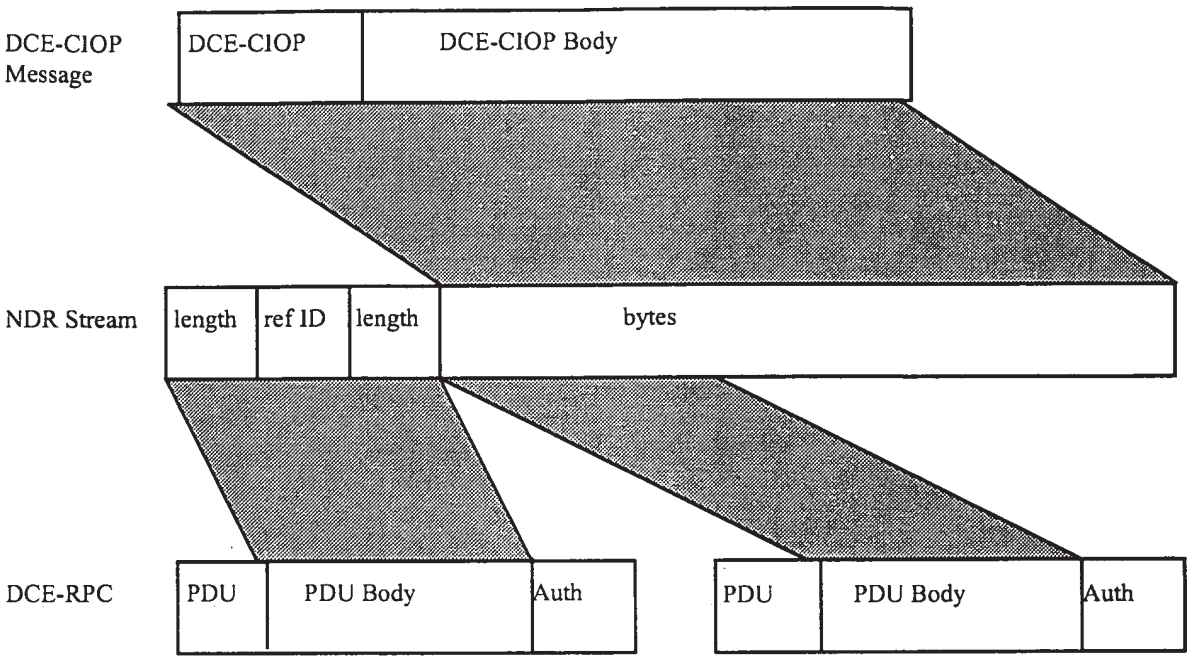

Figure 16-2 Array-based Interface Protocol Layering

The NDR stream, formed by concatenating the PDU bodies, is interpreted as the NDR representation of the DCE IDL **`message_type`** structure. The **`length`** member is encoded first, followed by the **`data`** member. The **`data`** member is a full pointer, which is represented in NDR as a referent ID. In this case, this non-NULL pointer is the first (and only) pointer to the referent, so the referent ID is 1 and it is followed by the representation of the referent. The referent is a conformant array of bytes, which is represented in NDR as an unsigned long indicating the length, followed by that number of bytes. The bytes form the DCE-CIOP message.

16.3.2.1 Invoke

The **`invoke`** RPC is used by a DCE-CIOP client process to attempt to invoke a CORBA operation in the server process identified by the **`binding_handle`** parameter. The **`request_message`** input parameter contains a DCE-CIOP invoke request message. The **`response_message`** output parameter returns a DCE-CIOP invoke response message from the server to the client.

16.3.2.2 Locate

The **locate** RPC is used by a DCE-CIOP client process to query the server process identified by the **binding_handle** parameter for the location of the server process where requests should be sent. The **request_message** and **response_message** parameters are used similarly to the parameters of the **invoke** RPC.

16.4 DCE-CIOP Message Formats

This section defines the message formats used by DCE-CIOP. These message formats are specified in OMG IDL, are encoded using CDR, and are transmitted over DCE-RPC as either pipes or arrays of bytes as described in Section 16.3, "DCE-CIOP Message Transport," on page 16-5.

16.4.1 DCE_CIOP Invoke Request Message

DCE-CIOP invoke request messages encode CORBA object requests, including attribute accessor operations and **CORBA::Object** operations such as **get_interface** and **get_implementation**. Invoke requests are passed from client to server as the **request_message** parameter of an **invoke** RPC.

A DCE-CIOP invoke request message is made up of a header and a body. The header has a fixed format, while the format of the body is determined by the operation's IDL definition.

16.4.1.1 Invoke request header

DCE-CIOP request headers have the following structure:

```
module DCE_CIOP {                                          // IDL
    struct InvokeRequestHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        sequence <octet> object_key;
        string operation;
        CORBA::Principal principal;

        // in and inout parameters follow
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **service_context** contains any ORB service data that needs to be sent from the client to the server.

- **object_key** contains opaque data used to identify the object that is the target of the operation[5]. Its value is obtained from the **object_key** field of the **TAG_INTERNET_IOP** profile or the **TAG_COMPLETE_OBJECT_KEY** component of the **TAG_MULTIPLE_COMPONENTS** profile.
- **operation** contains the name of the CORBA operation being invoked. The case of the operation name must match the case of the operation name specified in the OMG IDL source for the interface being used.

  Attribute accessors have names as follows:
  - Attribute selector: operation name is "_get_<attribute>"
  - Attribute mutator: operation name is "_set_<attribute>"

  CORBA::Object pseudo-operations have operation names as follows:
  - **get_interface** - operation name is "_interface"
  - **get_implementation** - operation name is "_implementation"
  - **is_a** - operation name is "_is_a"
  - **non_existent** - operation name is "_non_existent"
- **Principal** contains a value identifying the requesting principal. No particular meaning or semantics are associated with this value. It is provided to support the `BOA::get_principal` operation.

16.4.1.2 Invoke request body

The invoke request body contains the following items encoded in this order:

- All **in** and **inout** parameters, in the order in which they are specified in the operation's OMG IDL definition, from left to right.
- An optional Context pseudo object, encoded as described in Section 15.3.5.4, "Context," on page 15-29[6]. This item is only included if the operation's OMG IDL definition includes a context expression, and only includes context members as defined in that expression.

16.4.2 DCE-CIOP Invoke Response Message

Invoke response messages are returned from servers to clients as the `response_message` parameter of an `invoke` RPC.

---

5. Previous revisions of DCE-CIOP included an endpoint_id member, obtained from an optional TAG_ENDPOINT_ID component, as part of the object identity. The endpoint ID, if used, is now contained within the object key, and its position is specified by the optional TAG_ENDPOINT_ID_POSITION component.
6. Previous revisions of DCE-CIOP encoded the Context in the InvokeRequestHeader. It has been moved to the body for consistency with GIOP.

Like invoke request messages, an invoke response message is made up of a header and a body. The header has a fixed format, while the format of the body depends on the operation's OMG IDL definition and the outcome of the invocation.

16.4.2.1 Invoke response header

DCE-CIOP invoke response headers have the following structure:

```
module DCE_CIOP {                                          // IDL
    enum InvokeResponseStatus {
        INVOKE_NO_EXCEPTION,
        INVOKE_USER_EXCEPTION,
        INVOKE_SYSTEM_EXCEPTION,
        INVOKE_LOCATION_FORWARD,
        INVOKE_TRY_AGAIN
    };

    struct InvokeResponseHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        InvokeResponseStatus status;

        // if status = INVOKE_NO_EXCEPTION,
        // result then inouts and outs follow


        // if status = INVOKE_USER_EXCEPTION or
        // INVOKE_SYSTEM_EXCEPTION, an exception follows

        // if status = INVOKE_LOCATION_FORWARD, an
        // IOP::IOR follows
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **service_context** contains any ORB service data that needs to be sent from the client to the server.
- **status** indicates the completion status of the associated request, and also determines the contents of the body.

16.4.2.2 Invoke Response Body

The contents of the invoke response body depends on the value of the **`status`** member of the invoke response header, as well as the OMG IDL definition of the operation being invoked. Its format is one of the following:

- If the **status** value is **INVOKE_NO_EXCEPTION**, then the body contains the operation result value (if any), followed by all inout and out parameters, in the order in which they appear in the operation signature, from left to right.
- If the **status** value is **INVOKE_USER_EXCEPTION** or **INVOKE_SYSTEM_EXCEPTION**, then the body contains the exception, encoded as in GIOP.
- If the **status** value is **INVOKE_LOCATION_FORWARD**, then the body contains a new IOR containing a **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** profile whose components can be used to communicate with the object specified in the invoke request message[7]. This profile must provide at least one new DCE-CIOP binding component. The client ORB is responsible for resending the request to the server identified by the new profile. This operation should be transparent to the client program making the request. See "DCE-CIOP Object Location" on page 16-21 for more details.
- If the **status** value is **INVOKE_TRY_AGAIN**, then the body is empty and the client should reissue the `invoke` RPC, possibly after a short delay[8].

16.4.3 DCE-CIOP Locate Request Message

Locate request messages may be sent from a client to a server, as the `request_message` parameter of a `locate` RPC, to determine the following regarding a specified object reference:

- Whether the object reference is valid.
- Whether the current server is capable of directly receiving requests for the object reference.
- If not capable, to solicit an address to which requests for the object reference should be sent.

For details on the usage of the `locate` RPC, see Section 16.6, "DCE-CIOP Object Location," on page 16-21.

Locate request messages contain a fixed-format header, but no body.

16.4.3.1 Locate Request Header

DCE-CIOP locate request headers have the following format:

```
module DCE_CIOP {                    // IDL
    struct LocateRequestHeader {
        boolean byte_order;
```

---

7. Previous revisions of DCE-CIOP returned a MultipleComponentProfile structure. An IOR is now returned to allow either a TAG_INTERNET_IOP or a TAG_MULTIPLE_COMPONENTS profile to be used.
8. An exponential back-off algorithm is recommended, but not required.

```
        sequence <octet> object_key;
        string operation;

        // no body follows
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **object_key** contains opaque data used to identify the object that is the target of the operation. Its value is obtained from the **object_key** field of the **TAG_INTERNET_IOP** profile or the **TAG_COMPLETE_OBJECT_KEY** component of the **TAG_MULTIPLE_COMPONENTS** profile.
- **operation** contains the name of the CORBA operation being invoked. It is encoded as in the invoke request header.

16.4.4 DCE-CIOP Locate Response Message

Locate response messages are sent from servers to clients as the `response_message` parameter of a `locate` RPC. They consist of a fixed-format header, and a body whose format depends on information in the header.

16.4.4.1 Locate Response Header

DCE-CIOP locate response headers have the following format:

```
module DCE_CIOP {                          // IDL
    enum LocateResponseStatus {
        LOCATE_UNKNOWN_OBJECT,
        LOCATE_OBJECT_HERE,
        LOCATE_LOCATION_FORWARD,
        LOCATE_TRY_AGAIN
    };
    struct LocateResponseHeader {
        boolean byte_order;
        LocateResponseStatus status;

        // if status = LOCATE_LOCATION_FORWARD, an
        // IOP::IOR follows
    };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.

- **status** indicates whether the object is valid and whether it is located in this server. It determines the contents of the body.

16.4.4.2 Locate Response Body

The contents of the locate response body depends on the value of the `status` member of the locate response header. Its format is one of the following:

- If the `status` value is LOCATE_UNKNOWN_OBJECT, then the object specified in the corresponding locate request message is unknown to the server. The locate reply body is empty in this case.
- If the `status` value is LOCATE_OBJECT_HERE, then this server (the originator of the locate response message) can directly receive requests for the specified object. The locate response body is also empty in this case.
- If the `status` value is LOCATE_LOCATION_FORWARD, then the locate response body contains a new IOR containing a **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** profile whose components can be used to communicate with the object specified in the locate request message. This profile must provide at least one new DCE-CIOP binding component.
- If the status value is LOCATE_TRY_AGAIN, the locate response body is empty and the client should reissue the `locate` RPC, possibly after a short delay[9].

16.5 DCE-CIOP Object References

The information necessary to invoke operations on objects using DCE-CIOP is encoded in an IOR in a profile identified either by **TAG_INTERNET_IOP** or by **TAG_MULTIPLE_COMPONENTS**. The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in Section 15.7.2, "IIOP IOR Profiles," on page 15-51. The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures, described in Section 13.6, "An Information Model for Object References," on page 13-14.

DCE-CIOP defines a number of IOR components that can be included in either profile. Each is identified by a unique tag, and the encoding and semantics of the associated **component_data** are specified.

Either IOR profile can contain components for other protocols in addition to DCE-CIOP, and can contain components used by other kinds of ORB services. For example, an ORB vendor can define its own private components within this profile to support the vendor's native protocol. Several of the components defined for DCE-CIOP may be of use to other protocols as well. The following component descriptions will note whether

9. An exponential back-off algorithm is recommended, but not required.

the component is intended solely for DCE-CIOP or can be used by other protocols, whether the component is required or optional for DCE-CIOP, and whether more than one instance of the component can be included in a profile.

A conforming implementation of DCE-CIOP is only required to generate and recognize the components defined here. Unrecognized components should be preserved but ignored. Implementations should also be prepared to encounter profiles identified by **TAG_INTERNET_IOP** or by **TAG_MULTIPLE_COMPONENTS** that do not support DCE-CIOP.

16.5.1 DCE-CIOP String Binding Component

A DCE-CIOP string binding component, identified by **TAG_DCE_STRING_BINDING**, contains a fully or partially bound string binding. A string binding provides the information necessary for DCE-RPC to establish communication with a server process that can either service the client's requests itself, or provide the location of another process that can. The DCE API routine **`rpc_binding_from_string_binding`** can be used to convert a string binding to the DCE binding handle required to communicate with a server as described in Section 16.3, "DCE-CIOP Message Transport," on page 16-5.

This component is intended to be used only by DCE-CIOP. At least one string binding or binding name component must be present for an IOR profile to support DCE-CIOP.

Multiple string binding components can be included in a profile to define endpoints for different DCE protocols, or to identify multiple servers or agents capable of servicing the request.

The string binding component is defined as follows:

```
module DCE_CIOP {                                          \\ IDL
    const IOP::ComponentId TAG_DCE_STRING_BINDING = 100;
};
```

A **TaggedComponent** structure is built for the string binding component by setting the tag member to **TAG_DCE_STRING_BINDING** and setting the **component_data** member to the value of a DCE string binding. The string is represented directly in the sequence of octets, including the terminating NUL, without further encoding.

The format of a string binding is defined in Chapter 3 of the OSF *AES/Distributed Computing RPC Volume*. The DCE API function **`rpc_binding_from_string_binding`** converts a string binding into a binding handle that can be used by a client ORB as the first parameter to the **`invoke`** and **`locate`** RPCs.

A string binding contains:

- A protocol sequence
- A network address
- An optional endpoint

- An optional object UUID

DCE object UUIDs are used to identify server process endpoints, which can each support any number of CORBA objects. DCE object UUIDs do not necessarily correspond to individual CORBA objects.

A partially bound string binding does not contain an endpoint. Since the DCE-RPC run-time uses an endpoint mapper to complete a partial binding, and multiple ORB servers might be located on the same host, partially bound string bindings must contain object UUIDs to distinguish different endpoints at the same network address.

16.5.2 DCE-CIOP Binding Name Component

A DCE-CIOP binding name component is identified by **TAG_DCE_BINDING_NAME**. It contains a name that can be used with a DCE nameservice such as CDS or GDS to obtain the binding handle needed to communicate with a server process.

This component is intended for use only by DCE-CIOP. Multiple binding name components can be included to identify multiple servers or agents capable of handling a request. At least one binding name or string binding component must be present for a profile to support DCE-CIOP.

The binding name component is defined by the following OMG IDL:

```
module DCE_CIOP {                                                    // IDL
    const IOP::ComponentId TAG_DCE_BINDING_NAME = 101;

    struct BindingNameComponent {
        unsigned long entry_name_syntax;
        string entry_name;
        string object_uuid;
    };
};
```

A **TaggedComponent** structure is built for the binding name component by setting the tag member to **TAG_DCE_BINDING_NAME** and setting the **component_data member** to a CDR encapsulation of a **BindingNameComponent** structure.

16.5.2.1 BindingNameComponent

The **BindingNameComponent** structure contains the information necessary to query a DCE nameservice such as CDS. A client ORB can use the **entry_name_syntax, entry_name**, and **object_uuid** members of the **BindingName** structure with the `rpc_ns_binding_import_*` or `rpc_ns_binding_lookup_*` families of DCE API routines to obtain binding handles to communicate with a server. If the `object_uuid` member is an empty string, a nil object UUID should be passed to these DCE API routines.

16.5.3 DCE-CIOP No Pipes Component

The optional component identified by **TAG_DCE_NO_PIPES** indicates to an ORB client that the server does not support the `dce_ciop_pipe` DCE-RPC interface. It is only a hint, and can be safely ignored. As described in Section 16.3, "DCE-CIOP Message Transport," on page 16-5, the client must fall back to the array-based interface if the pipe-based interface is not available in the server.

```
module DCE_CIOP {
    const IOP::ComponentId TAG_DCE_NO_PIPES = 102;
};
```

A **TaggedComponent** structure with a `tag` member of **TAG_DCE_NO_PIPES** must have an empty **component_data** member.

This component is intended for use only by DCE-CIOP, and a profile should not contain more than one component with this tag.

16.5.4 Complete Object Key Component

An IOR profile supporting DCE-CIOP must include an object key that identifies the object the IOR represents. The object key is an opaque sequence of octets used as the **object_key** member in invoke and locate request message headers. In a **TAG_INTERNET_IOP** profile, the **object_key** member of the **IIOP::ProfileBody_1_1** structure is used. In a **TAG_MULTIPLE_COMPONENTS** profile supporting DCE-CIOP[10], a single **TAG_COMPLETE_OBJECT_KEY** component must be included to identify the object.

The **TAG_COMPLETE_OBJECT_KEY** component is available for use by all protocols that use the **TAG_MULTIPLE_COMPONENTS** profile. By sharing this component, protocols can avoid duplicating object identity information. This component should never be included in a **TAG_INTERNET_IOP** profile.

```
module IOP {                                                    // IDL
    const ComponentId TAG_COMPLETE_OBJECT_KEY = 5;
};
```

The sequence of octets comprising the **component_data** of this component is not interpreted by the client process. Its format only needs to be understood by the server process and any location agent that it uses.

10. Previous DCE-CIOP revisions used a different component.

16.5.5 Endpoint ID Position Component

An optional endpoint ID position component can be included in IOR profiles to enable client ORBs to minimize resource utilization and to avoid redundant locate messages. It can be used by other protocols as well as by DCE-CIOP. No more than one endpoint ID position component can be included in a profile.

```
module IOP {                                                    // IDL
    const ComponentId TAG_ENDPOINT_ID_POSITION = 6;

    struct EndpointIdPositionComponent {
        unsigned short begin;
        unsigned short end;
    };
};
```

An endpoint ID position component, identified by **TAG_ENDPOINT_ID_POSITION**, indicates the portion of the profile's object key that identifies the endpoint at which operations on an object can be invoked. The **component_data** is a CDR encapsulation of an **EndpointIdPositionComponent** structure. The **begin** member of this structure specifies the index in the object key of the first octet of the endpoint ID. The **end** member specifies the index of the last octet of the endpoint ID. An index value of zero specifies the first octet of the object key. The value of **end** must be greater than the value of **begin**, but less than the total number of octets in the object key. The endpoint ID is made up of the octets located between these two indices inclusively.

The endpoint ID should be unique within the domain of interoperability. A binary or stringified UUID is recommended.

If multiple objects have the same endpoint ID, they can be messaged to at a single endpoint, avoiding the need to locate each object individually. DCE-CIOP clients can use a single binding handle to invoke requests on all of the objects with a common endpoint ID. See Section 16.6.4, "Use of the Location Policy and the Endpoint ID," on page 16-24.

16.5.6 Location Policy Component

An optional location policy component can be included in IOR profiles to specify when a DCE-CIOP client ORB should perform a `locate` RPC before attempting to perform an `invoke` RPC. No more than one location policy component should be included in a profile, and it can be used by other protocols that have location algorithms similar to DCE-CIOP.

```
module IOP {                                                    // IDL
    const ComponentId TAG_LOCATION_POLICY = 12;

    // IDL does not support octet constants
    #define LOCATE_NEVER = 0
    #define LOCATE_OBJECT = 1
```

```
    #define LOCATE_OPERATION = 2
    #define LOCATE_ALWAYS = 3
};
```

A **TaggedComponent** structure for a location policy component is built by setting the tag member to **TAG_LOCATION_POLICY** and setting the **component_data** member to a sequence containing a single octet, whose value is **LOCATE_NEVER**, **LOCATE_OBJECT**, **LOCATE_OPERATION**, or **LOCATE_ALWAYS**.

If a location policy component is not present in a profile, the client should assume a location policy of **LOCATE_OBJECT**.

A client should interpret the location policy as follows:

- **LOCATE_NEVER** - Perform only the **invoke** RPC. No **locate** RPC is necessary.
- **LOCATE_OBJECT** - Perform a **locate** RPC once per object. The **operation** member of the locate request message will be ignored.
- **LOCATE_OPERATION** - Perform a separate **locate** RPC for each distinct operation on the object. This policy can be used when different methods of an object are located in different processes.
- **LOCATE_ALWAYS** - Perform a separate **locate** RPC for each invocation on the object. This policy can be used to support server-per-method activation.

The location policy is a hint that enables a client to avoid unnecessary **locate** RPCs and to avoid **invoke** RPCs that return **INVOKE_LOCATION_FORWARD** status. It is not needed to provide correct semantics, and can be ignored. Even when this hint is utilized, an **invoke** RPC might result in an **INVOKE_LOCATION_FORWARD** response. See Section 16.6, "DCE-CIOP Object Location," on page 16-21 for more details.

A client does not need to implement all location policies to make use of this hint. A location policy with a higher value can be substituted for one with a lower value. For instance, a client might treat **LOCATE_OPERATION** as **LOCATE_ALWAYS** to avoid having to keep track of binding information for each operation on an object.

When combined with an endpoint ID component, a location policy of **LOCATE_OBJECT** indicates that the client should perform a **locate** RPC for the first object with a particular endpoint ID, and then just perform an **invoke** RPC for other objects with the same endpoint ID. When a location policy of **LOCATE_NEVER** is combined with an endpoint ID component, only **invoke** RPCs need be performed. The **LOCATE_ALWAYS** and **LOCATE_OPERATION** policies should not be combined with an endpoint ID component in a profile.

16.6 DCE-CIOP Object Location

This section describes how DCE-CIOP client ORBs locate the server ORBs that can perform operations on an object via the **invoke** RPC.

16.6.1 Location Mechanism Overview

DCE-CIOP is defined to support object migration and location services without dictating the existence of specific ORB architectures or features. The protocol features are based on the following observations:

- A given transport address does not necessarily correspond to any specific ORB architectural component (such as an object adapter, server process, ORB process, locator, etc.). It merely implies the existence of some agent to which requests may be sent.
- The "agent" (receiver of an RPC) may have one of the following roles with respect to a particular object reference:
  - The agent may be able to accept object requests directly for the object and return replies. The agent may or may not own the actual object implementation; it may be a gateway that transforms the request and passes it on to another process or ORB. From DCE-CIOP's perspective, it is only important that invoke request messages can be sent directly to the agent.
  - The agent may not be able to accept direct requests for any objects, but acts instead as a location service. Any invoke request messages sent to the agent would result in either exceptions or replies with **INVOKE_LOCATION_FORWARD** status, providing new addresses to which requests may be sent. Such agents would also respond to locate request messages with appropriate locate response messages.
  - The agent may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.
  - The agent may directly respond to requests for a particular object at one point in time, and provide a forwarding location at a later time.
- Server ORBs are not required to implement location forwarding mechanisms. An ORB can be implemented with the policy that servers either support direct access to an object, or return exceptions. Such a server ORB would always return locate response messages with either **LOCATE_OBJECT_HERE** or **LOCATE_UNKNOWN_OBJECT** status, and never **LOCATE_LOCATION_FORWARD** status. It would also never return invoke response messages with **INVOKE_LOCATION_FORWARD** status.
- Client ORBs must, however, be able to accept and process invoke response messages with **INVOKE_LOCATION_FORWARD** status, since any server ORB may choose to implement a location service. Whether a client ORB chooses to send locate request messages is at the discretion of the client.
- Client ORBs that send locate request messages can use the location policy component found in DCE-CIOP IOR profiles to decide whether to send a locate request message before sending an invoke request message. See Section 16.5.6, "Location Policy Component," on page 16-20. This hint can be safely ignored by a client ORB.

- A client should not make any assumptions about the longevity of addresses returned by location forwarding mechanisms. If a binding handle based on location forwarding information is used successfully, but then fails, subsequent attempts to send requests to the same object should start with the original address specified in the object reference.

In general, the use of location forwarding mechanisms is at the discretion of ORBs, available to be used for optimization and to support flexible object location and migration behaviors.

16.6.2 Activation

Activation of ORB servers is transparent to ORB clients using DCE-CIOP. Unless an IOR refers to a transient object, the agent addressed by the IOR profile should either be permanently active, or should be activated on demand by DCE's endpoint mapper.

The current DCE endpoint mapper, rpcd, does not provide activation. In ORB server environments using rpcd, the agent addressed by an IOR must not only be capable of locating the object, it must also be able to activate it if necessary. A future DCE endpoint mapper may provide automatic activation, but client ORB implementations do not need to be aware of this distinction.

16.6.3 Basic Location Algorithm

ORB clients can use the following algorithm to locate the server capable of handling the `invoke` RPC for a particular operation:

1. Pick a profile with **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** from the IOR. Make this the *original* profile and the *current* profile. If no profiles with either tag are available, operations cannot be invoked using DCE-CIOP with this IOR.

2. Get a binding handle to try from the *current* profile. See Section 16.5.1, "DCE-CIOP String Binding Component," on page 16-17 and Section 16.5.2, "DCE-CIOP Binding Name Component," on page 16-18. If no binding handles can be obtained, the server cannot be located using the *current* profile, so go to step 1.

3. Perform either a `locate` or `invoke` RPC using the object key from the *current* profile.
   - If the RPC fails, go to step 2 to try a different binding handle.
   - If the RPC returns **INVOKE_TRY_AGAIN** or **LOCATE_TRY_AGAIN**, try the same RPC again, possibly after a delay.
   - If the RPC returns either **INVOKE_LOCATION_FORWARD** or **LOCATE_LOCATION_FORWARD**, make the new IOR profile returned in the response message body the *current* profile and go to step 2.
   - If the RPC returns **LOCATE_UNKNOWN_OBJECT**, and the *original* profile was used, the object no longer exists.
   - Otherwise, the server has been successfully located.

Any **invoke RPC** might return **INVOKE_LOCATION_FORWARD**, in which case the client ORB should make the returned profile the *current* profile, and re-enter the location algorithm at step 2.

If an RPC on a binding handle fails after it has been used successfully, the client ORB should start over at step 1.

16.6.4 Use of the Location Policy and the Endpoint ID

The algorithm above will allow a client ORB to successfully locate a server ORB, if possible, so that operations can be invoked using DCE-CIOP. But unnecessary **locate** RPCs may be performed, and **invoke** RPCs may be performed when **locate** RPCs would be more efficient. The optional location policy and endpoint ID position components can be used by the client ORB, if present in the IOR profile, to optimize this algorithm.

16.6.4.1 Current location policy

The client ORB can decide whether to perform a **locate** RPC or an **invoke** RPC in step 3 based on the location policy of the *current* IOR profile. If the *current* profile has a **TAG_LOCATION_POLICY** component with a value of **LOCATE_NEVER**, the client should perform an **invoke** RPC. Otherwise, it should perform a **locate** RPC.

16.6.4.2 Original location policy

The client ORB can use the location policy of the *original* IOR profile as follows to determine whether it is necessary to perform the location algorithm for a particular invocation:

- **LOCATE_OBJECT** or **LOCATE_NEVER** - A binding handle previously used successfully to invoke an operation on an object can be reused for all operations on the same object. The client only needs to perform the location algorithm once per object.
- **LOCATE_OPERATION** - A binding handle previously used successfully to invoke an operation on an object can be reused for that same operation on the same object. The client only needs to perform the location algorithm once per operation.
- **LOCATE_ALWAYS** - Binding handles should not be reused. The client needs to perform the location algorithm once per invocation.

16.6.4.3 Original Endpoint ID

If a component with **TAG_ENDPOINT_ID_POSITION** is present in the *original* IOR profile, the client ORB can reuse a binding handle that was successfully used to perform an operation on another object with the same endpoint ID. The client only needs to perform the location algorithm once per endpoint.

An endpoint ID position component should never be combined in the same profile with a location policy of **LOCATE_OPERATION** or **LOCATE_ALWAYS**.

16.7 OMG IDL for the DCE CIOP Module

This section shows the **DCE_CIOP** module and **DCE_CIOP** additions to the **IOP** module.

```
module DCE_CIOP {
    struct InvokeRequestHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        sequence <octet> object_key;
        string operation;
        CORBA::Principal principal;

        // in and inout parameters follow
    };

    enum InvokeResponseStatus {
        INVOKE_NO_EXCEPTION,
        INVOKE_USER_EXCEPTION,
        INVOKE_SYSTEM_EXCEPTION,
        INVOKE_LOCATION_FORWARD,
        INVOKE_TRY_AGAIN
    };
    struct InvokeResponseHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        InvokeResponseStatus status;

        // if status = INVOKE_NO_EXCEPTION,
        // result then inouts and outs follow

        // if status = INVOKE_USER_EXCEPTION or
        // INVOKE_SYSTEM_EXCEPTION, an exception follows

        // if status = INVOKE_LOCATION_FORWARD, an
        // IOP::IOR follows
    };
    struct LocateRequestHeader {
        boolean byte_order;
        sequence <octet> object_key;
        string operation;

        // no body follows
    };

enum LocateResponseStatus {
    LOCATE_UNKNOWN_OBJECT,
    LOCATE_OBJECT_HERE,
    LOCATE_LOCATION_FORWARD,
    LOCATE_TRY_AGAIN
```

```
};
struct LocateResponseHeader {
    boolean byte_order;
    LocateResponseStatus status;

    // if status = LOCATE_LOCATION_FORWARD, an
    // IOP::IOR follows
};

const IOP::ComponentId TAG_DCE_STRING_BINDING = 100;

const IOP::ComponentId TAG_DCE_BINDING_NAME = 101;

struct BindingNameComponent {
        unsigned long entry_name_syntax;
        string entry_name;
        string object_uuid;
    };

    const IOP::ComponentId TAG_DCE_NO_PIPES = 102;
};

module IOP {
    const ComponentId TAG_COMPLETE_OBJECT_KEY = 5;

    const ComponentId TAG_ENDPOINT_ID_POSITION = 6;

    struct EndpointIdPositionComponent {
        unsigned short begin;
        unsigned short end;
    };

    const ComponentId TAG_LOCATION_POLICY = 12;

    // IDL does not support octet constants
    #define LOCATE_NEVER 0
    #define LOCATE_OBJECT 1
    #define LOCATE_OPERATION 2
    #define LOCATE_ALWAYS 3
};
```

16.8 References for this Chapter

AES/Distributed Computing RPC Volume, P T R Prentice Hall, Englewood Cliffs, New Jersey, 1994

CAE Specification C309 X/Open DCE: Remote Procedure Call, X/Open Company Limited, Reading, UK

Mapping: COM and CORBA 18

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.

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 $-2^{15}...2^{15} - 1$
long	long	long	Signed integer with a range of $-2^{31}...2^{31} - 1$
unsigned short	unsigned short	unsigned short	Unsigned integer with a range of $0...2^{16} - 1$
unsigned long	unsigned long	unsigned long	Unsigned integer with a range of $0...2^{32} - 1$
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 ml;
            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 ... Tl;
    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 callee[1]. 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 InsufFunds { 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_InsufFunds
      {
      float balance;
      };

struct Bank_InvalidAmount
      {
      float amount;
      };

struct BANK_Account_NotAuthorized
      {
      };

interface IBANK_AccountUserExceptions : IUnknown
      {
   HRESULT _get_InsufFunds( [out]  BANK_InsufFunds
                     * 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.

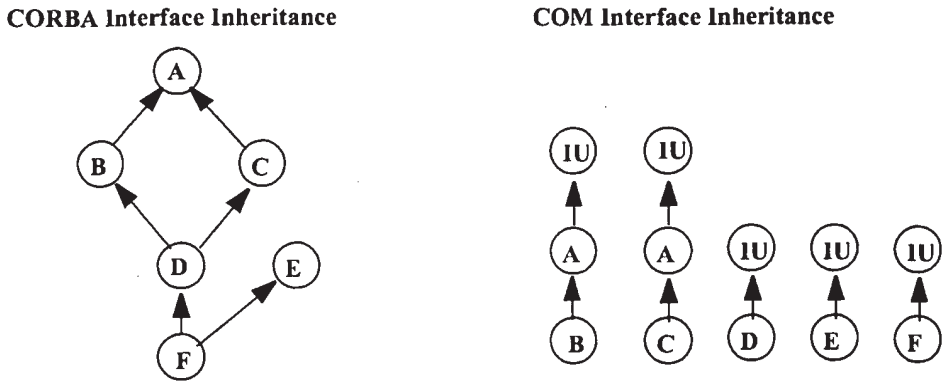

Figure 18-1 CORBA Interface Inheritance to COM Interface Inheritance Mapping

```
//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 proceeded 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;
} CORBATypeCodeExceptions;

typedef LPTSTR      RepositoryId;
typedef LPTSTR      Identifier;

typedef [v1_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,
```