General Inter-ORB Protocol 15

This chapter specifies a General Inter-ORB Protocol (GIOP) for ORB interoperability, which can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions. This chapter also defines a specific mapping of the GIOP, which runs directly over TCP/IP connections, called the Internet Inter-ORB Protocol (IIOP). The IIOP must be supported by conforming networked ORB products regardless of other aspects of their implementation. Such support does not require using it internally; conforming ORBs may also provide bridges to this protocol.

Contents

This chapter contains the following sections.

15.1 Goals of the General Inter-ORB Protocol

The GIOP and IIOP support protocol-level ORB interoperability in a general, low-cost manner. The following objectives were pursued vigorously in the GIOP design:

- ***Widest possible availability*** - The GIOP and IIOP are based on the most widely-used and flexible communications transport mechanism available (TCP/IP), and defines the minimum additional protocol layers necessary to transfer CORBA requests between ORBs.
- ***Simplicity*** - The GIOP is intended to be as simple as possible, while meeting other design goals. Simplicity is deemed the best approach to ensure a variety of independent, compatible implementations.
- ***Scalability*** - The GIOP/IIOP protocol should support ORBs, and networks of bridged ORBs, to the size of today's Internet, and beyond.
- ***Low cost*** - Adding support for GIOP/IIOP to an existing or new ORB design should require small engineering investment. Moreover, the run-time costs required to support IIOP in deployed ORBs should be minimal.
- ***Generality*** - While the IIOP is initially defined for TCP/IP, GIOP message formats are designed to be used with any transport layer that meets a minimal set of assumptions; specifically, the GIOP is designed to be implemented on other connection-oriented transport protocols.
- ***Architectural neutrality*** - The GIOP specification makes minimal assumptions about the architecture of agents that will support it. The GIOP specification treats ORBs as opaque entities with unknown architectures.

The approach a particular ORB takes to providing support for the GIOP/IIOP is undefined. For example, an ORB could choose to use the IIOP as its internal protocol, it could choose to externalize IIOP as much as possible by implementing it in a half-bridge, or it could choose a strategy between these two extremes. All that is required of a conforming ORB is that some entity or entities in, or associated with, the ORB be able to send and receive IIOP messages.

15.2 GIOP Overview

The GIOP specification consists of the following elements:

- *The Common Data Representation (CDR) definition*. CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for "on-the-wire" transfer between ORBs and Inter-ORB bridges (agents).
- *GIOP Message Formats*. GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.
- *GIOP Transport Assumptions*. The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

The IIOP specification adds the following element to the GIOP specification:

- *Internet IOP Message Transport*. The IIOP specification describes how agents open TCP/IP connections and use them to transfer GIOP messages.

The IIOP is not a separate specification; it is a specialization, or mapping, of the GIOP to a specific transport (TCP/IP). The GIOP specification (without the transport-specific IIOP element) may be considered as a separate conformance point for future mappings to other transport layers.

The complete OMG IDL specifications for the GIOP and IIOP are shown in Section 15.10, “OMG IDL,” on page 15-60. Fragments of the specification are used throughout this chapter as necessary.

15.2.1 Common Data Representation (CDR)

CDR is a transfer syntax, mapping from data types defined in OMG IDL to a bicanonical, low-level representation for transfer between agents. CDR has the following features:

- ***Variable byte ordering*** - Machines with a common byte order may exchange messages without byte swapping. When communicating machines have different byte order, the message originator determines the message byte order, and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.
- ***Aligned primitive types*** - Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory.
- ***Complete OMG IDL Mapping*** - CDR describes representations for all OMG IDL data types, including transferable pseudo-objects such as TypeCodes. Where necessary, CDR defines representations for data types whose representations are undefined or implementation-dependent in the CORBA Core specifications.

15.2.2 GIOP Message Overview

The GIOP specifies formats for messages that are exchanged between inter-operating ORBs. GIOP message formats have the following features:

- ***Few, simple messages.*** With only seven message formats, the GIOP supports full CORBA functionality between ORBs, with extended capabilities supporting object location services, dynamic migration, and efficient management of communication resources. GIOP semantics require no format or binding negotiations. In most cases, clients can send requests to objects immediately upon opening a connection.
- ***Dynamic object location.*** Many ORBs’ architectures allow an object implementation to be activated at different locations during its lifetime, and may allow objects to migrate dynamically. GIOP messages provide support for object location and migration, without requiring ORBs to implement such mechanisms when unnecessary or inappropriate to an ORB’s architecture.

- ***Full CORBA support*** - GIOP messages directly support all functions and behaviors required by CORBA, including exception reporting, passing operation context, and remote object reference operations (such as **CORBA::Object::get_interface**).

GIOP also supports passing service-specific context, such as the transaction context defined by the Transaction Service (the Transaction Service is described in *CORBAservices: Common Object Service Specifications*). This mechanism is designed to support any service that requires service related context to be implicitly passed with requests.

15.2.3 GIOP Message Transfer

The GIOP specification is designed to operate over any connection-oriented transport protocol that meets a minimal set of assumptions (described in Section 15.5, “GIOP Message Transport,” on page 15-46). GIOP uses underlying transport connections in the following ways:

- ***Asymmetrical connection usage*** - The GIOP defines two distinct roles with respect to connections, client, and server. The client side of a connection originates the connection, and sends object requests over the connection. In GIOP versions 1.0 and 1.1, the server side receives requests and sends replies. The server side of a connection may not send object requests. This restriction, which was included to make GIOP 1.0 and 1.1 much simpler and avoid certain race conditions, has been relaxed for GIOP version 1.2 and 1.3, as specified in the BiDirectional GIOP specification, see Section 15.8, “Bi-Directional GIOP,” on page 15-56.
- ***Request multiplexing*** - If desirable, multiple clients within an ORB may share a connection to send requests to a particular ORB or server. Each request uniquely identifies its target object. Multiple independent requests for different objects, or a single object, may be sent on the same connection.
- ***Overlapping requests*** - In general, GIOP message ordering constraints are minimal. GIOP is designed to allow overlapping asynchronous requests; it does not dictate the relative ordering of requests or replies. Unique request/reply identifiers provide proper correlation of related messages. Implementations are free to impose any internal message ordering constraints required by their ORB architectures.
- ***Connection management*** - GIOP defines messages for request cancellation and orderly connection shutdown. These features allow ORBs to reclaim and reuse idle connection resources.

15.3 CDR Transfer Syntax

The Common Data Representation (CDR) transfer syntax is the format in which the GIOP represents OMG IDL data types in an octet stream.

An octet stream is an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine over some IPC mechanism or network transport. For the purposes of this discussion, an octet stream is an arbitrarily long (but finite) sequence of eight-bit values (octets) with a well-defined beginning. The octets in the stream are numbered from *0* to *n-1*, where *n* is the size of the stream. The numeric position of an octet in the stream is called its *index*. Octet indices are used to calculate alignment boundaries, as described in Section 15.3.1.1, “Alignment,” on page 15-5.

GIOP defines two distinct kinds of octet streams, messages and encapsulations. Messages are the basic units of information exchange in GIOP, described in detail in Section 15.4, “GIOP Message Formats,” on page 15-30.

Encapsulations are octet streams into which OMG IDL data structures may be marshaled independently, apart from any particular message context. Once a data structure has been encapsulated, the **octet** stream can be represented as the OMG IDL opaque data type **sequence<octet>**, which can be marshaled subsequently into a message or another encapsulation. Encapsulations allow complex constants (such as TypeCodes) to be pre-marshaled; they also allow certain message components to be handled without requiring full unmarshaling. Whenever encapsulations are used in CDR or the GIOP, they are clearly noted.

15.3.1 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see Section 15.4, “GIOP Message Formats,” on page 15-30) include tags in message headers that indicate the byte ordering in the message. Encapsulations include an initial flag that indicates the byte ordering within the encapsulation, described in Section 15.3.3, “Encapsulation,” on page 15-14. The byte ordering of any encapsulation may be different from the message or encapsulation within which it is nested. It is the responsibility of the message recipient to translate byte ordering if necessary. Primitive data types are encoded in multiples of octets. An **octet** is an 8-bit value.

15.3.1.1 Alignment

In order to allow primitive data to be moved into and out of octet streams with instructions specifically designed for those primitive data types, in CDR all primitive data types must be aligned on their natural boundaries (i.e., the alignment boundary of a primitive datum is equal to the size of the datum in **octets**). Any primitive of size *n* octets must start at an octet stream index that is a multiple of *n*. In CDR, *n* is one of 1, 2, 4, or 8.

Where necessary, an alignment gap precedes the representation of a primitive datum. The value of **octets** in alignment gaps is undefined. A gap must be the minimum size necessary to align the following primitive. Table 15-1 gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 15-1 Alignment requirements for OMG IDL primitive data types

TYPE	OCTET ALIGNMENT
char	1
wchar	1, 2 or 4 for GIOP 1.1 | 1 for GIOP 1.2 and 1.3
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long double	8
boolean	1
enum	4

Alignment is defined above as being relative to the beginning of an octet stream. The first octet of the stream is octet index zero (0); any data type may be stored starting at this index. Such octet streams begin at the start of a GIOP message header (see Section 15.4.1, "GIOP Message Header," on page 15-31) and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See Section 15.3.3, "Encapsulation," on page 15-14).

15.3.1.2 Integer Data Types

Figure 15-1 on page 15-7 illustrates the representations for OMG IDL integer data types, including the following data types:

- **short**
- **unsigned short**
- **long**
- **unsigned long**
- **long long**
- **unsigned long long**

The figure illustrates bit ordering and size. Signed types (**short**, **long**, and **long long**) are represented as two's complement numbers; unsigned versions of these types are represented as unsigned binary numbers.

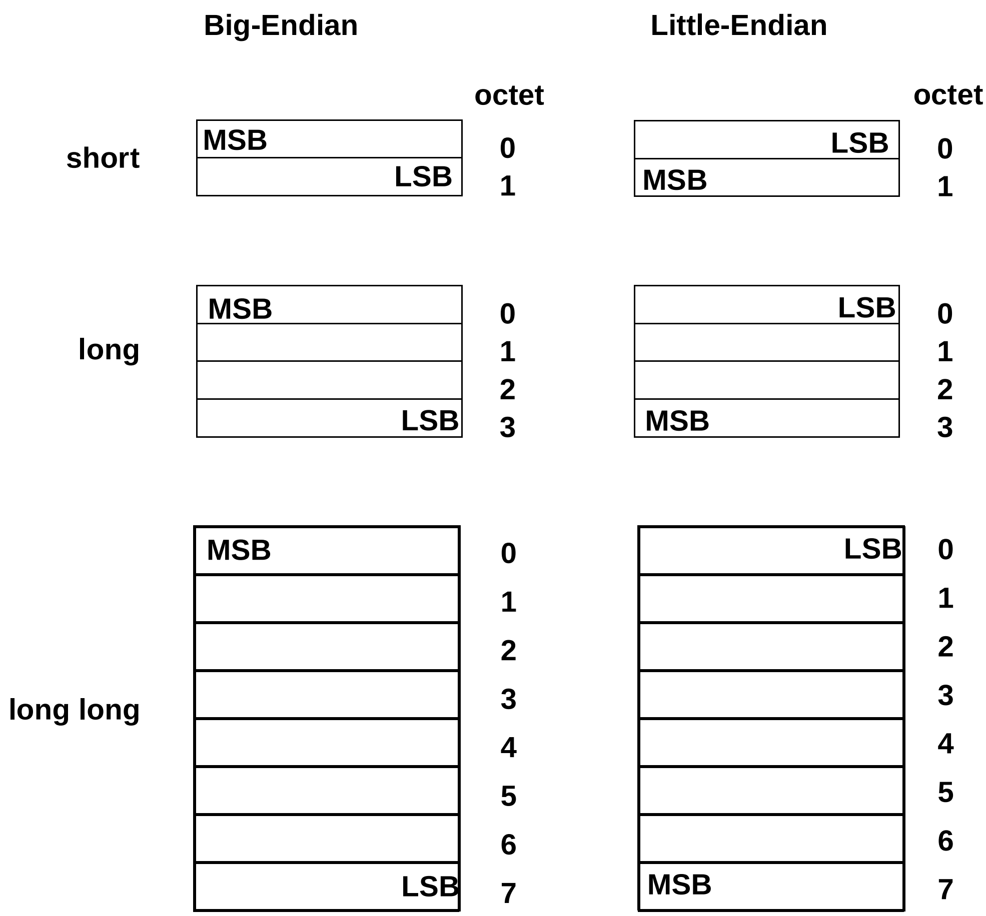

Figure 15-1 Sizes and bit ordering in big-endian and little-endian encodings of OMG IDL integer data types, both signed and unsigned.

15.3.1.3 *Floating Point Data Types*

Figure 15-2 on page 15-9 illustrates the representation of floating point numbers. These exactly follow the IEEE standard formats for floating point numbers[1], selected parts of which are abstracted here for explanatory purposes. The diagram shows three different components for floating points numbers, the sign bit (s), the exponent (e) and the fractional part (f) of the mantissa. The sign bit has values of 0 or 1, representing positive and negative numbers, respectively.

1. "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

For single-precision float values the exponent is 8 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 127. The fractional mantissa (f1 - f3) is a 23-bit value f where 1.0 <= f < 2.0, f1 being most significant and f3 being least significant. The value of a normalized number is described by:

$$-1^{sign} \times 2^{(exponent - 127)} \times (1 + fraction)$$

For double-precision values the exponent is 11 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 1023. The fractional mantissa (f1 - f7) is a 52-bit value m where 1.0 <= m < 2.0, f1 being most significant and f7 being least significant. The value of a normalized number is described by:

$$-1^{sign} \times 2^{(exponent - 1023)} \times (1 + fraction)$$

For double-extended floating-point values the exponent is 15 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are the most significant. The fractional mantissa (f1 through f14) is 112 bits long, with f1 being the most significant. The value of a **long double** is determined by:

$$-1^{sign} \times 2^{(exponent - 16383)} \times (1 + fraction)$$

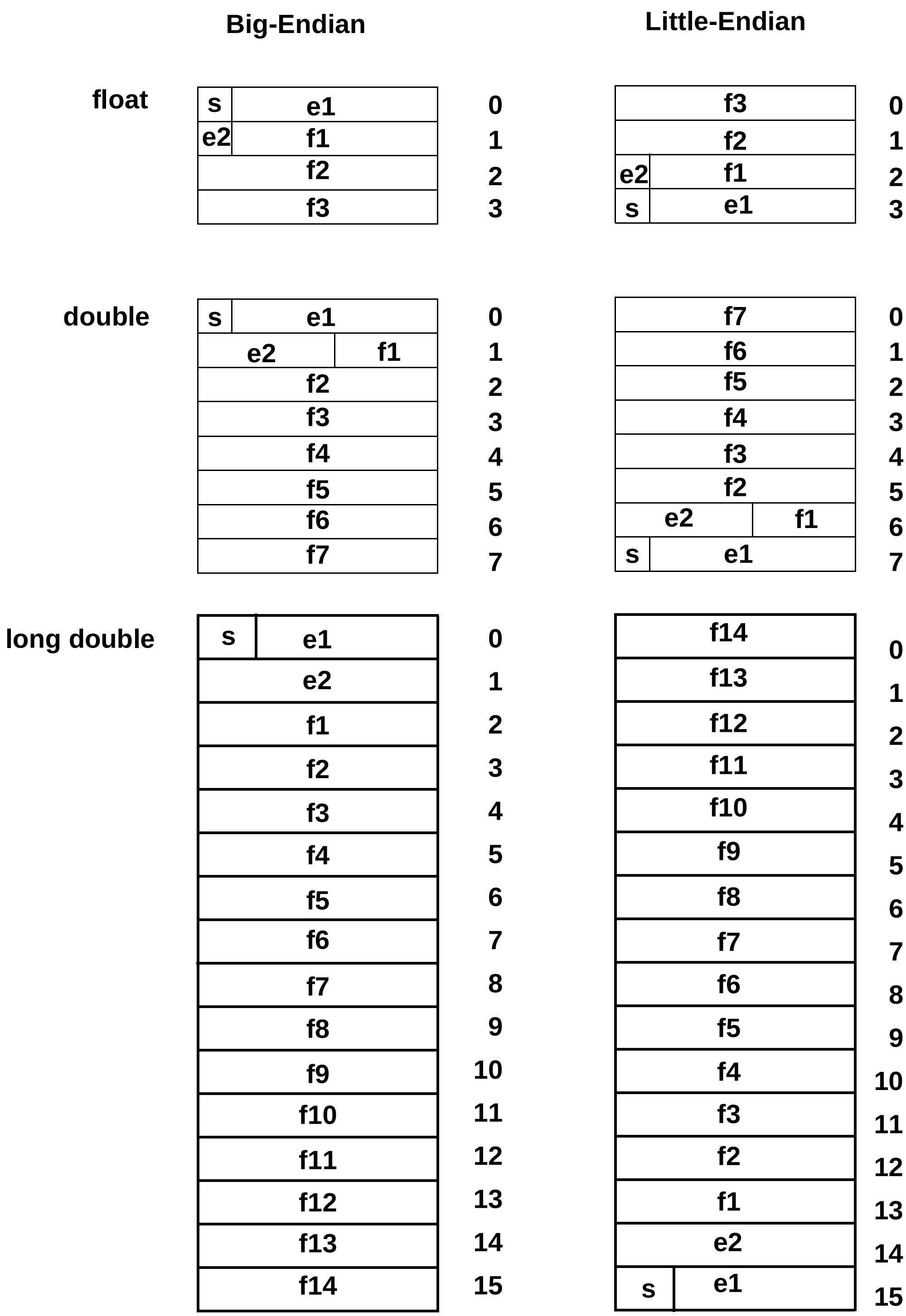

Figure 15-2 Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single, double precision, and double extended floating point numbers.

15.3.1.4 Octet

Octets are uninterpreted 8-bit values whose contents are guaranteed not to undergo any conversion during transmission. For the purposes of describing possible **octet** values in this specification, octets may be considered as unsigned 8-bit integer values.

15.3.1.5 Boolean

Boolean values are encoded as single octets, where **TRUE** is the value 1, and **FALSE** as 0.

15.3.1.6 Character Types

An IDL character is represented as a single octet; the code set used for transmission of character data (e.g., TCS-C) between a particular client and server ORBs is determined via the process described in Section 13.10, “Code Set Conversion,” on page 13-37. In the case of multi-byte encodings of characters, a single instance of the **char** type may only hold one octet of any multi-byte character encoding.

Note – Full representation of multi-byte characters will require the use of an array of IDL **char** variables.

For GIOP version 1.1, the transfer syntax for an IDL wide character depends on whether the transmission code set (TCS-W, which is determined via the process described in Section 13.10, “Code Set Conversion,” on page 13-37) is byte-oriented or non-byte-oriented:

- Byte-oriented (e.g., SJIS). Each wide character is represented as one or more octets, as defined by the selected TCS-W.
- Non-byte-oriented (e.g., Unicode UTF-16). Each wide character is represented as one or more codepoints. A codepoint is the same as “Coded-Character data element,” or “CC data element” in ISO terminology. Each codepoint is encoded using a fixed number of bits as determined by the selected TCS-W. The OSF Character and Code Set Registry may be examined using the interfaces in Section 13.10.5, “Relevant OSFM Registry Interfaces,” on page 13-50 to determine the maximum length (max_bytes) of any character codepoint. For example, if the TCS-W is ISO 10646 UCS-2 (Universal Character Set containing 2 bytes), then wide characters are represented as **unsigned shorts**. For ISO 10646 UCS-4, they are represented as **unsigned longs**.

For GIOP version 1.2, and 1.3 **wchar** is encoded as an unsigned binary octet value, followed by the elements of the octet sequence representing the encoded value of the **wchar**. The initial octet contains a count of the number of elements in the sequence, and the elements of the sequence of octets represent the **wchar**, using the negotiated wide character encoding.

Note – The GIOP 1.2 and 1.3 encoding of **wchar** is similar to the encoding of an octet sequence, except for its use of a single octet to encode the value of the length.

For GIOP versions prior to 1.2 and 1.3, interoperability for wchar is limited to the use of two- octet fixed-length encoding.

Wchar values in encapsulations are assumed to be encoded using GIOP version 1.2 and 1.3 CDR.

If UTF-16 is selected as the TCS-W the CDR encoding purposes can be big endian or little endian, but defaults to big endian. By placing a BOM (byte order marker) at the front of the wstring or wchar encoding, it can be sent either big-endian or little-endian. In particular, the CDR rules for endian-ness of UTF-16 encoded wstring or wchar values are as follows:

- If the first two bytes (after the length indication) are FE FF, it's big-endian.
- If the first two bytes (after the length indication) are FF FE, it's little-endian.
- If the first two bytes (after the length indication) are neither, it's big-endian.

If an ORB decides to use BOM to indicate endianness, it shall add the BOM to the beginning of wchar or wstring values when encoding the value, since it is not present in wchar or wstring values passed by the user.

If a BOM is present at the beginning of a wchar or wstring received in a GIOP message, the ORB shall remove the BOM before passing the value to the user.

If a client orb erroneously sends **wchar** or **wstring** data in a GIOP 1.0 message, the server shall generate a MARSHAL standard system exception, with standard minor code 5.

If a server erroneously sends **wchar** data in a GIOP 1.0 response, the client ORB shall raise a MARSHAL exception to the client application with standard minor code 6.

15.3.2 OMG IDL Constructed Types

Constructed types are built from OMG IDL’s data types using facilities defined by the OMG IDL language.

15.3.2.1 Alignment

Constructed types have no alignment restrictions beyond those of their primitive components. The alignment of those primitive types is not intended to support use of marshaling buffers as equivalent to the implementation of constructed data types within any particular language environment. GIOP assumes that agents will usually construct structured data types by copying primitive data between the marshaled buffer and the appropriate in-memory data structure layout for the language mapping implementation involved.

15.3.2.2 Struct

The components of a structure are encoded in the order of their declaration in the structure. Each component is encoded as defined for its data type.

15.3.2.3 Union

Unions are encoded as the discriminant tag of the type specified in the union declaration, followed by the representation of any selected member, encoded as its type indicates.

15.3.2.4 Array

Arrays are encoded as the array elements in sequence. As the array length is fixed, no length values are encoded. Each element is encoded as defined for the type of the array. In multidimensional arrays, the elements are ordered so the index of the first dimension varies most slowly, and the index of the last dimension varies most quickly.

15.3.2.5 Sequence

Sequences are encoded as an unsigned long value, followed by the elements of the sequence. The initial unsigned long contains the number of elements in the sequence. The elements of the sequence are encoded as specified for their type.

15.3.2.6 Enum

Enum values are encoded as unsigned longs. The numeric values associated with enum identifiers are determined by the order in which the identifiers appear in the enum declaration. The first enum identifier has the numeric value zero (0). Successive enum identifiers take ascending numeric values, in order of declaration from left to right.

15.3.2.7 Strings and Wide Strings

A string is encoded as an **unsigned long** indicating the length of the string in octets, followed by the string value in single- or multi-byte form represented as a sequence of octets. The string contents include a single terminating null character. The string length includes the null character, so an empty string has a length of 1.

For GIOP version 1.1, 1.2, and 1.3, when encoding a string, always encode the length as the total number of bytes used by the encoding string, regardless of whether the encoding is byte-oriented or not.

For GIOP version 1.1, a wide string is encoded as an **unsigned long** indicating the length of the string in octets or unsigned integers (determined by the transfer syntax for wchar) followed by the individual wide characters. The string contents include a single terminating null character. The string length includes the null character. The terminating null character for a wstring is also a wide character.

For GIOP version 1.2 and 1.3, when encoding a **wstring**, always encode the length as the total number of octets used by the encoded value, regardless of whether the encoding is byte-oriented or not. For GIOP version 1.2 and 1.3 a **wstring** is not terminated by a null character. In particular, in GIOP version 1.2 and 1.3 a length of 0 is legal for **wstring**.

Note – For GIOP versions prior to 1.2 and 1.3, interoperability for **wstring** is limited to the use of two-octet fixed-length encoding.

Wstring values in encapsulations are assumed to be encoded using GIOP version 1.2 and 1.3 CDR.

15.3.2.8 Fixed-Point Decimal Type

The IDL **fixed** type has no alignment restrictions, and is represented as shown in Table 15-4 on page 15-14. Each **octet** contains (up to) two decimal digits. If the **fixed** type has an odd number of decimal digits, then the representation begins with the first (most significant) digit — d0 in the figure. Otherwise, this first half-octet is all zero, and the first digit is in the second half-octet — d1 in the figure. The sign configuration, in the last half-octet of the representation, is 0xD for negative numbers and 0xC for positive and zero values.

The number of digits present must equal the number of significant digits specified in the IDL definition for the fixed type being marshalled, with the exception of the inclusion of a leading 0x0 half octet when there are an even number of significant digits.

Decimal digits are encoded as hexadecimal values in each half-octet as follows:

Decimal Digit	Half-Octet Value
0	0x0
1	0x1
2	0x2
...	...
9	0x9

Figure 15-3 Decimal Digit Encoding for Fixed Type

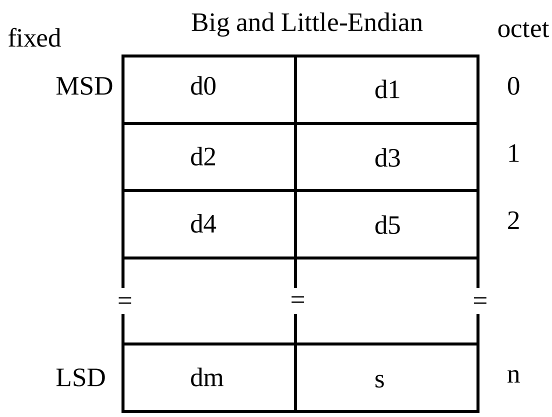

Figure 15-4 IDL Fixed Type Representation

15.3.3 Encapsulation

As described above, OMG IDL data types may be independently marshaled into encapsulation octet streams. The octet stream is represented as the OMG IDL type **sequence<octet>**, which may be subsequently included in a GIOP message or nested in another encapsulation.

The GIOP and IIOP explicitly use encapsulations in three places: *TypeCodes* (see Section 15.3.5.1, “TypeCode,” on page 15-23), the IIOP protocol profile inside an IOR (see Section 15.3.6, “Object References,” on page 15-30), and in service-specific context (see Section 13.7, “Service Context,” on page 13-28). In addition, some ORBs may choose to use an encapsulation to hold the **object_key** (see Section 15.7.2, “IIOP IOR Profiles,” on page15-52), or in other places that a **sequence<octet>** data type is in use.

When encapsulating OMG IDL data types, the first octet in the stream (index 0) contains a boolean value indicating the byte ordering of the encapsulated data. If the value is **FALSE** (0), the encapsulated data is encoded in big-endian order; if **TRUE** (1), the data is encoded in little-endian order, exactly like the byte order flag in GIOP message headers (see Section 15.4.1, “GIOP Message Header,” on page 15-31). This value is not part of the data being encapsulated, but is part of the octet stream holding the encapsulation. Following the byte order flag, the data to be encapsulated is marshaled into the buffer as defined by CDR encoding rules. Marshaled data are aligned relative to the beginning of the octet stream (the first octet of which is occupied by the byte order flag).

When the encapsulation is encoded as type **sequence<octet>** for subsequent marshaling, an unsigned long value containing the sequence length is prefixed to the octet stream, as prescribed for sequences (see Section 15.3.2.5, “Sequence,” on page 15-12). The length value is not part of the encapsulation’s octet stream, and does not affect alignment of data within the encapsulation.

Note that this guarantees a four-octet alignment of the start of all encapsulated data within GIOP messages and nested encapsulations.[2]

Whenever the use of an encapsulation is specified, the GIOP version to use for encoding the encapsulation, if different than GIOP version 1.0, shall be explicitly defined (i.e., the default is GIOP 1.0).

If a parameter with IDL char or string type is defined to be carried in an encapsulation using GIOP version greater than 1.0, the transmission Code Set for characters (TCS-C), to be used when encoding the encapsulation, shall also be explicitly defined.

If a parameter with IDL wchar or wstring type is defined to be carried in an encapsulation using GIOP version greater than 1.0, the transmission Code Set for wide characters (TCS-W), to be used when encoding the encapsulation shall also be explicitly defined.

15.3.4 Value Types

Value types are built from OMG IDL's value type definitions. Their representation and encoding is defined in this section.

Value types may be used to transmit and encode complex state. The general approach is to support the transmission of the data (state) and type information encoded as **RepositoryID**s.

The loading (and possible transmission) of code is outside of the scope of the GIOP definition, but enough information is carried to support it, via the CodeBase object.

The format makes a provision for the support of custom marshaling (i.e., the encoding and transmission of state using application-defined code). Consistency between custom encoders and decoders is not ensured by the protocol

The encoding supports all of the features of value types as well as supporting the "chunking" of value types. It does so in a compact way.

At a high level the format can be described as the linearization of a graph. The graph is the depth-first exploration of the transitive closure that starts at the top-level value object and follows its "reference to value objects" fields (an ordinary remote reference is just written as an IOR). It is a recursive encoding similar to the one used for TypeCodes. An indirection is used to point to a value that has already been encoded.

The data members are written beginning with the highest possible base type to the most derived type in the order of their declaration.

2. Accordingly, in cases where encapsulated data holds data with natural alignment of greater than four octets, some processors may need to copy the octet data before removing it from the encapsulation. For example, an appropriate way to deal with long long discriminator type in an encapsulation for a union TypeCode is to encode the body of the encapsulation as if it was aligned at the 8 byte boundary, and then copy the encoded value into the encapsulation. This may result in long long data values inside the encapsulation being aligned on only a 4 byte boundary when viewed from outside the encapsulation.

15.3.4.1 Partial Type Information and Versioning

The format provides support for partial type information and versioning issues in the receiving context. However the encoding has been designed so that this information is only required when “advanced features” such as truncation are used.

The presence (or absence) of type information and codebase URL information is indicated by flags within the <value_tag>, which is a **long** in the range between **0x7fffff00** and **0x7fffffff** inclusive. The last octet of this tag is interpreted as follows:

- The least significant bit (<value_tag> & **0x00000001**) is the value **1** if a <codebase_URL> is present. If this bit is **0**, no <codebase_URL> follows in the encoding. The <codebase_URL> is a blank-separated list of one or more URLs.
- The second and third least significant bits (<value_tag> & **0x00000006**) are:
  - the value **0** if no type information is present in the encoding. This indicates the actual parameter is the same type as the formal argument.
  - the value **2** if only a single repository id is present in the encoding, which indicates the most derived type of the actual parameter (which may be either the same type as the formal argument or one of its derived types).
  - the value **6** if the partial type information list of repository ids is present in the encoding as a list of repository ids.

When a list of **RepositoryIDs** is present, the encoding is a **long** specifying the number of **RepositoryIDs**, followed by the **RepositoryIDs**. The first **RepositoryID** is the id for the most derived type of the value. If this type has any base types, the sending context is responsible for listing the **RepositoryIDs** for all the base types to which it is safe to truncate the value passed. These truncatable base types are listed in order, going up the derivation hierarchy. The sending context may choose to (but need not) terminate the list at any point after it has sent a **RepositoryID** for a type well-known to the receiving context. A well-known type is any of the following:

- a type that is a formal parameter, result of the method call, or exception, for which this GIOP message is being marshaled
- a base type of a well-known type
- a member type of a well-known type
- an element type of a well known type

For value types that have an RMI: **RepositoryId**, ORBs must include at least the most derived **RepositoryId**, in the value type encoding.

For value types marshaled as abstract interfaces (see Section 15.3.7, “Abstract Interfaces,” on page 15-30), **RepositoryId** information must be included in the value type encoding.

If the receiving context needs more typing information than is contained in a GIOP message that contains a codebase URL information, it can go back to the sending context and perform a lookup based on that **RepositoryID** to retrieve more typing information (e.g., the type graph).

CORBA **RepositoryIDs** may contain standard version identification (major and minor version numbers or a hash code information). The ORB run time may use this information to check whether the version of the value being transmitted is compatible with the version expected. In the event of a version mismatch, the ORB may apply product-specific truncation/conversion rules (with the help of a local interface repository or the **SendingContext::RunTime** service). For example, the Java serialization model of truncation/conversion across versions can be supported. See the JDK 1.1 documentation for a detailed specification of this model.

15.3.4.2 Example

The following examples demonstrate legal combinations of truncatability, actual parameter types and GIOP encodings. This is not intended to be an exhaustive list of legal possibilities.

The following example uses valuetypes **animal** and **horse**, where **horse** is derived from **animal**. The actual parameters passed to the specified operations are **an_animal** of runtime type **animal** and **a_horse** of runtime type **horse**.

The following combinations of truncatability, actual parameter types and GIOP encodings are legal.

1. If there is a single operation:

op1(in animal a);

a) If the type **horse** cannot be truncated to **animal** (i.e., **horse** is declared):

valuetype horse: animal ...

then the encoding is as shown below:

Actual Invocation	Legal Encoding
op1(a_horse)	2 horse
	6 1 horse

Note that if the type horse is not available to the receiver, then the receiver throws a demarshaling exception.

b). If the type **horse** can be truncated to **animal** (i.e., **horse** is declared):

valuetype horse: truncatable animal ...

then the encoding is as shown below

Actual Invocation	Legal Encoding
op1(a_horse)	6 2 horse animal

Note that if the type horse is not available to the receiver, then the receiver tries to truncate to animal.

c) Regardless of the truncation relationships, when the exact type of the formal argument is sent, then the encoding is as shown below:

Actual Invocation	Legal Encoding
op1(an_animal)	0
	2 animal
	6 1 animal

2. Given the additional operation:

op2(in horse h);

(i.e., the sender knows that both types **horse** and **animal** and their derivation relationship are known to the receiver)

a). If the type horse cannot be truncated to animal (i.e., horse is declared):

valuetype horse: animal ...

then the encoding is as shown below:

Actual Invocation	Legal Encoding
op2(a_horse)	2 horse
	6 1 horse

Note that the demarshaling exception of case 1 will not occur, since horse is available to the receiver.

b). If the type horse can be truncated to animal (i.e., horse is declared):

valuetype horse: truncatable animal ...

then the encoding is as shown below:

Actual Invocation	Legal Encoding
op2 (a_horse)	2 horse
	6 1 horse
	6 2 horse animal

Note that truncation will not occur, since horse is available to the receiver.

15.3.4.3 *Scope of the Indirections*

The special value **0xffffffff** introduces an indirection (i.e., it directs the decoder to go somewhere else in the marshaling buffer to find what it is looking for). This can be codebase URL information that has already been encoded, a **RepositoryID** that has already been encoded, a list of repository IDs that has already been encoded, or another value object that is shared in a graph. **0xffffffff** is always followed by a **long** indicating where to go in the buffer. A repositoryID or URL, which is the target of an indirection used for encoding a valuetype must have been introduced as the type or codebase information for a valuetype.

It is not permissible for a repositoryID marshalled for some purpose other than as the type information of a valuetype to use indirection to reference a previously marshaled value. The encoding used to indicate an indirection is the same as that used for recursive TypeCodes (i.e., a **0xffffffff** indirection marker followed by a **long** offset (in units of **octets**) from the beginning of the long offset). As an example, this means that an offset of negative four (-4) is illegal, because it is self-indirecting to its indirection marker. Indirections may refer to any preceding location in the GIOP message, including previous fragments if fragmentation is used. This includes any previously marshaled parameters. Non-negative offsets are reserved for future use. Indirections may not cross encapsulation boundaries.

Fragmentation support in GIOP versions 1.1, 1.2, and 1.3 introduces the possibility of a header for a **FragmentMessage** being marshaled between the target of an indirection and the start of the encapsulation containing the indirection. The octets occupied by any such headers are not included in the calculation of the offset value.

15.3.4.4 *Null Values*

All value types have a distinguished "null." All null values are encoded by the <null_tag> (0x0). The CDR encoding of null values includes no type information.

15.3.4.5 *Other Encoding Information*

A "new" value is coded as a value header followed by the value's state. The header contains a tag and codebase URL information if appropriate, followed by the **RepositoryID** and an octet flag of bits. Because the same **RepositoryID** (and codebase URL information) could be repeated many times in a single request when sending a complex graph, they are encoded as a regular string the first time they appear, and use an indirection for later occurrences.

15.3.4.6 *Fragmentation*

It is anticipated that value types may be rather large, particularly when a graph is being transmitted. Hence the encoding supports the breaking up of the serialization into an arbitrary number of chunks in order to facilitate incremental processing.

Values with truncatable base types need a length indication in case the receiver needs to truncate them to a base type. Value types that are custom marshaled also need a length indication so that the ORB run time can know exactly where they end in the stream without relying on user-defined code. This allows the ORB to maintain consistency and ensure the integrity of the GIOP stream when the user-written custom marshaling and demarshaling does not marshal the entire value state. For simplicity of encoding, we use a length indication for all values whether or not they have a truncatable base type or use custom marshaling.

If limited space is available for marshaling, it may be necessary for the ORB to send the contents of a marshaling buffer containing a partially marshaled value as a GIOP fragment. At that point in the marshaling, the length of the entire value being marshaled may not be known. Calculating this length may require processing as costly as marshaling the entire value. It is therefore desirable to allow the value to be encoded as multiple chunks, each with its own length. This allows the portion of a value that occupies a marshaling buffer to be sent as a chunk of known length with no need for additional length calculation processing.

The data may be split into multiple chunks at arbitrary points except within primitive CDR types, arrays of primitive types, strings, and wstrings, or between the tag and offset of indirections. It is never necessary to end a chunk within one of these types as the length of these types is known before starting to marshal them so they can be added to the length of the currently open chunk. It is the responsibility of the CDR stream to hide the chunking from the marshaling code.

The presence (or absence) of chunking is indicated by flags within the <value_tag>. The fourth least significant bit (<value_tag> & **0x00000008**) is the value 1 if a chunked encoding is used for the value's state. The chunked encoding is required for custom marshaling and truncation. If this bit is 0, the state is encoded as <**octets**>.

Each chunk is preceded by a positive long, which specifies the number of octets in the chunk.

A chunked value is terminated by an end tag that is a non-positive long so the start of the next value can be differentiated from the start of another chunk. In the case of values that contain other values (e.g., a linked list) the "nested" value is started without there being an end tag. The absolute value of an end tag (when it finally appears) indicates the nesting level of the value being terminated. A single end tag can be used to terminate multiple nested values. The detailed rules are as follows:

- End tags, chunk size tags, and value tags are encoded using non-overlapping ranges so that the unmarshaling code can tell after reading each chunk whether:
  - another chunk follows (positive tag).
  - one or multiple value types are ending at a given point in the stream (negative tag).
  - a nested value follows (special large positive tag).
- The end tag is a negative long whose value is the negation of the absolute nesting depth of the value type ending at this point in the CDR stream. Any value types that have not already been ended and whose nesting depth is greater than the depth indicated by the end tag are also implicitly ended. The end tag value **0** is reserved

for future use (e.g., supporting a nesting depth of more than **2^31**). The outermost value type will always be terminated by an end tag with a value of **-1**. Enclosing non-chunked valuetypes are not considered when determining the nesting depth.

The following example describes how end tags may be used. Consider a valuetype declaration that contains two member values:

```
// IDL
    valuetype simpleNode{ ... };
    valuetype node truncatable simpleNode {
    public node node1;
    public node node2;
};
```

When an instance of type '**node**' is passed as a parameter of type '**simpleNode**' a chunked encoding is used. In all cases, the outermost value is terminated with an end tag with a value of **-1**. The nested value '**node1**' is terminated with an end tag with a value of **-2** since only the second-level value '**node1**' ends at that point. Since the nested value '**node2**' coterminates with the outermost value, either of the following end tag layouts is legal:

- A single end tag with a value of **-1** marks the termination of the outermost value, implying the termination of the nested value, '**node2**'as well. This is the most compact marshaling.
- An end tag with a value of **-2** marks the termination of the nested value, '**node2**.' This is then followed by an end tag with a value of **-1** to mark the termination of the outermost value.

Because data members are encoded in their declaration order, declaring a value type data member of a value type last is likely to result in more compact encoding on the wire because it maximizes the number of values ending at the same place and so allows a single end tag to be used for multiple values. The canonical example for that is a linked list.

- For the purposes of chunking, values encoded as indirections or null are treated as non-value data.
- Chunks are never nested. When a value is nested within another value, the outer value's chunk ends at the place in the stream where the inner value starts. If the outer value has non-value data to be marshaled following the inner value, the end tag for the inner value is followed by a continuation chunk for the remainder of the outer value.
- Regardless of the above rules, any value nested within a chunked value is always chunked. Furthermore, any such nested value that is truncatable must encode its type information as a list of **RepositoryIDs** (see Section 15.3.4.1, "Partial Type Information and Versioning," on page 15-16).

Truncating a value type in the receiving context may require keeping track of unused nested values (only during unmarshaling) in case further indirection tags point back to them. These values can be held in their "raw" GIOP form, as fully unmarshaled value objects, or in any other product-specific form.

Value types that are custom marshaled are encoded as chunks in order to let the ORB run-time know exactly where they end in the stream without relying on user-defined code.

15.3.4.7 Notation

The on-the-wire format is described by a BNF grammar with conventions similar to the ones used to define IDL syntax. ***The terminals of the grammar are to be interpreted differently.*** We are describing a protocol format. Although the terminals have the same names as IDL tokens they represent either:

- constant tags, or
- the GIOP CDR encoding of the corresponding IDL construct.

For example, **long** is a shorthand for the GIOP encoding of the IDL **long** data type - with all the GIOP alignment rules. Similarly **struct** is a shorthand for the GIOP CDR encoding of a **struct**.

A **(type) constant** means that an instance of the given type having the given value is encoded according to the rules for that type. So that **(long) 0** means that a CDR encoding for a long having the value **0** appears at that location.

15.3.4.8 The Format

```
(1)               <value>  ::= <value_tag> [ <codebase_URL> ]
                               [ <type_info> ] <state>
                             | <value_ref>
(2)           <value_ref>  ::= <indirection_tag> <indirection> | <null_tag>
(3)           <value_tag>  ::= long// 0x7fffff00 <= value_tag <= 0x7fffffff
(4)           <type_info>  ::= <rep_ids> | <repository_id>
(5)               <state>  ::= <octets> |<value_data>* [ <end_tag> ]
(6)          <value_data>  ::= <value_chunk> | <value>
(7)             <rep_ids>  ::= long <repository_id>+
                             | <indirection_tag> <indirection>
(8)       <repository_id>  ::= string | <indirection_tag> <indirection>
(9)         <value_chunk>  ::= <chunk_size_tag> <octets>
(10)           <null_tag>  ::= (long) 0
(11)    <indirection_tag>  ::= (long) 0xffffffff
(12)       <codebase_URL>  ::= string | <indirection_tag> <indirection>
(13)     <chunk_size_tag>  ::= long
                               //  0 < chunk_size_tag < 2^31-256 (0x7fffff00)
(14)            <end_tag>  ::= long     // -2^31 < end_tag < 0
(15)        <indirection>  ::= long     // -2^31 < indirection < 0
(16)             <octets>  ::= octet | octet <octets>
```

The concatenated octets of consecutive value chunks within a value encode state members for the value according to the following grammar:

```
(1)    <state members>  ::= <state_member>
                          |   <state_member> <state members>
(2)    <state_member>   ::= <value_ref>
                              // All legal IDL types should be here
                          |   octet
                          |   boolean
                          |   char
                          |   short
                          |   unsigned short
                          |   long
                          |   unsigned long
                          |   float
                          |   wchar
                          |   wstring
                          |   string
                          |   struct
                          |   union
                          |   sequence
                          |   array
                          |   Object
                          |   any
                          |   long long
                          |   unsigned long long
                          |   double
                          |   long double
                          |   fixed
```

15.3.5 Pseudo-Object Types

CORBA defines some kinds of entities that are neither primitive types (integral or floating point) nor constructed ones.

15.3.5.1 TypeCode

In general, TypeCodes are encoded as the **TCKind** enum value, potentially followed by values that represent the TypeCode parameters. Unfortunately, **TypeCodes** cannot be expressed simply in OMG IDL, since their definitions are recursive. The basic TypeCode representations are given in Table 15-2 on page 15-25. The *integer value* column of this table gives the **TCKind** enum value corresponding to the given TypeCode, and lists the parameters associated with such a TypeCode. The rest of this section presents the details of the encoding.

Basic TypeCode Encoding Framework

The encoding of a TypeCode is the **TCKind** enum value (encoded, like all **enum** values, using four octets), followed by zero or more parameter values. The encodings of the parameter lists fall into three general categories, and differ in order to conserve space and to support efficient traversal of the binary representation:

- Typecodes with an *empty parameter list* are encoded simply as the corresponding **TCKind** enum value.
- Typecodes with *simple parameter lists* are encoded as the **TCKind** enum value followed by the parameter value(s), encoded as indicated in Table 15-2. A "simple" parameter list has a fixed number of fixed length entries, or a single parameter that has its length encoded first.
- All other typecodes have *complex parameter lists*, which are encoded as the **TCKind** enum value followed by a CDR encapsulation octet sequence (see Section 15.3.3, "Encapsulation," on page 15-14) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 15-2.

The third column of Table 15-2 shows whether each parameter list is *empty*, *simple*, or *complex*. Also, note that an internal indirection facility is needed to represent some kinds of typecodes; this is explained in "Indirection: Recursive and Repeated TypeCodes" on page15-28. This indirection does not need to be exposed to application programmers.

TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 15-2 on page 15-25. The ordering and meaning of parameters is a superset of those given in Section 4.11, "TypeCodes," on page 4-52; more information is needed by CDR's representation in order to provide the full semantics of TypeCodes as shown by the API.

- Each parameter is written in the form *type (name)*, where *type* describes the parameter's type, and *name* describes the parameter's meaning.
- The encoding of some parameter lists (specifically, **tk_struct, tk_union, tk_enum**, and **tk_except**) contain a counted sequence of tuples.

  Such counted tuple sequences are written in the form *count {parameters}*, where *count* is the number of tuples in the encoded form, and the *parameters* enclosed in braces are available in each tuple instance. First the *count*, which is an `unsigned long`, and then each *parameter* in each tuple (using the noted type), is encoded in the CDR representation of the typecode. Each tuple is encoded, first parameter followed by second, before the next tuple is encoded (first, then second, etc.).

Note that the tuples identifying **struct**, union, **exception**, and **enum** members must be in the order defined in the OMG IDL definition text. Also, that the types of discriminant values in encoded **tk_union** TypeCodes are established by the second encoded parameter (*discriminant type*), and cannot be specified except with reference to a specific OMG IDL definition.[3]

3. This means that, for example, two OMG IDL unions that are textually equivalent, except that one uses a "char" discriminant, and the other uses a "long" one, would have different size encoded TypeCodes.

*Table 15-2*TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
tk_null	0	empty	– none –
tk_void	1	empty	– none –
tk_short	2	empty	– none –
tk_long	3	empty	– none –
tk_ushort	4	empty	– none –
tk_ulong	5	empty	– none –
tk_float	6	empty	– none –
tk_double	7	empty	– none –
tk_boolean	8	empty	– none –
tk_char	9	empty	– none –
tk_octet	10	empty	– none –
tk_any	11	empty	– none –
tk_TypeCode	12	empty	– none –
tk_Principal	13	empty	– none –
tk_objref	14	complex	string (repository ID), string(name)
tk_struct	15	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
tk_union	16	complex	string (repository ID), string(name), TypeCode (discriminant type), long (default used), ulong (count) {*discriminant type*[1] (label value), string (member name), TypeCode (member type)}
tk_enum	17	complex	string (repository ID), string (name), ulong (count) {string (member name)}

Table 15-2 TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
tk_string	18	simple	ulong (max length[2])
tk_sequence	19	complex	TypeCode (element type), ulong (max length[3])
tk_array	20	complex	TypeCode (element type), ulong (length)
tk_alias	21	complex	string (repository ID), string (name), TypeCode
tk_except	22	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
tk_longlong	23	empty	– none –
tk_ulonglong	24	empty	– none –
tk_longdouble	25	empty	– none –
tk_wchar	26	empty	– none –
tk_wstring	27	simple	ulong(max length or zero if unbounded)
tk_fixed	28	simple	ushort(digits), short(scale)
tk_value	29	complex	string (repository ID), string (name, may be empty), short(ValueModifier), TypeCode(of concrete base)[4], ulong (count), {string (member name), TypeCode (member type), short(Visibility)}
tk_value_box	30	complex	string (repository ID), string(name), TypeCode
tk_native	31	complex	string (repository ID), string(name)
tk_abstract_interface	32	complex	string(RepositoryId), string(name)
tk_local_interface	33	complex	string(RepositoryId), string(name)

Table 15-2 TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
tk_component	34	complex	string (repository ID), string(name)
tk_home	35	complex	string (repository ID), string(name)
tk_event	36	complex	string (repository ID), string (name, may be empty), short(ValueModifier), TypeCode(of concrete base)[5], ulong (count), {string (member name), TypeCode (member type), short(Visibility)}
– none –	0xffffffff	simple	long (indirection[6])

1. The type of union label values is determined by the second parameter, discriminant type.

2. For unbounded strings, this value is zero.

3. For unbounded sequences, this value is zero.

4. Should be **tk_null** if there is no concrete base.

5. Should be **tk_null** if there is no concrete base.

6. See “Indirection: Recursive and Repeated TypeCodes” on page 15-28.

Encoded Identifiers and Names

The Repository ID parameters in **tk_objref**, **tk_struct**, **tk_union**, **tk_enum**, **tk_alias**, **tk_except**, **tk_native**, **tk_value**, **tk_value_box** and **tk_abstract_interface** TypeCodes are Interface Repository **RepositoryId** values, whose format is described in the specification of the Interface Repository.

For GIOP 1.2 onwards, repositoryID values are required to be sent, if known by the ORB[4]. For GIOP 1.2 and 1.3 an empty repositoryID string is only allowed if a repositoryID value is not available to the ORB sending the type code.

For GIOP 1.0 and 1.1, **RepositoryId** values are required for **tk_objref** and **tk_except** TypeCodes; for **tk_struct**, **tk_union**, **tk_enum**, and **tk_alias** TypeCodes **RepositoryIds** are optional and encoded as empty strings if omitted.

4. A type code passed via a GIOP 1.2 connection shall contain non-empty repositoryID strings, unless a repositoryID value is not available to the sending ORB for a specific type code. This situation can arise, for example, if an ORB receives a type code containing empty repository IDs via a GIOP 1.0 or 1.1 connection and passes that type code on via a GIOP 1.2 connection).

The name parameters in **tk_objref, tk_struct, tk_union, tk_enum, tk_alias, tk_value, tk_value_box, tk_abstract_interface, tk_native** and **tk_except** TypeCodes and the member name parameters in **tk_struct, tk_union, tk_enum**, **tk_value** and **tk_except** TypeCodes are not specified by (or significant in) GIOP. Agents should not make assumptions about type equivalence based on these name values; only the structural information (including **RepositoryId** values, if provided) is significant. If provided, the strings should be the simple, unscoped names supplied in the OMG IDL definition text. If omitted, they are encoded as empty strings.

When a reference to a base **Object** is encoded, there are two allowed encodings for the Repository ID: either "**IDL:omg.org/CORBA/Object:1.0**" or "" may be used.

Encoding the tk_union Default Case

In **tk_union** TypeCodes, the **long** default used value is used to indicate which tuple in the sequence describes the union's default case. If this value is less than zero, then the union contains no default case. Otherwise, the value contains the zero-based index of the default case in the sequence of tuples describing union members.

The discriminant value used in the actual typecode parameter associated with the default member position in the list, may be any valid value of the discriminant type, and has no semantic significance (i.e., it should be ignored and is only included for syntactic completeness of union type code marshaling).

TypeCodes for Multi-Dimensional Arrays

The **tk_array** TypeCode only describes a single dimension of any array. TypeCodes for multi-dimensional arrays are constructed by nesting **tk_array** TypeCodes within other **tk_array** TypeCodes, one per array dimension. The outermost (or top-level) **tk_array** TypeCode describes the leftmost array index of the array as defined in IDL; the innermost nested **tk_array** TypeCode describes the rightmost index.

Indirection: Recursive and Repeated TypeCodes

The typecode representation of OMG IDL data types that can indirectly contain instances of themselves (e.g., **struct foo {sequence <foo> bar;}**) must also contain an indirection. Such an indirection is also useful to reduce the size of encodings; for example, unions with many cases sharing the same value.

CDR provides a constrained indirection to resolve this problem:

- The indirection applies only to TypeCodes nested within some "top-level" TypeCode. Indirected TypeCodes are not "freestanding," but only exist inside some other encoded TypeCode.
- Only the second (and subsequent) references to a TypeCode in that scope may use the indirection facility. The first reference to that TypeCode must be encoded using the normal rules. In the case of a recursive TypeCode, this means that the first instance will not have been fully encoded before a second one must be completely encoded.

The indirection is a numeric octet offset within the scope of the "top-level" TypeCode and points to the **TCKind** value for the typecode. (Note that the byte order of the **TCKind** value can be determined by its encoded value.) This indirection may well cross encapsulation boundaries, but this is not problematic because of the first constraint identified above. Because of the second constraint, the value of the offset will always be negative.

Fragmentation support in GIOP versions 1.1, 1.2, and 1.3 introduces the possibility of a header for a **FragmentMessage** being marshaled between the target of an indirection and the start of the encapsulation containing the indirection. The octets occupied by any such headers are not included in the calculation of the offset value.

The encoding of such an indirection is as a TypeCode with a "**TCKind** value" that has the special value 2^{32}-1 (**0xffffffff**, all ones). Such typecodes have a single (simple) parameter, which is the **long** offset (in units of octets) from the simple parameter. (This means that an offset of negative four (-4) is illegal because it will be self-indirecting.)

15.3.5.2 Any

Any values are encoded as a TypeCode (encoded as described above) followed by the encoded value. For **Any** values containing a **tk_null** or **tk_void TypeCode**, the encoded value shall have zero length (i.e., shall be absent).

15.3.5.3 Principal

Principal pseudo objects are encoded as **sequence<octet>**. In the absence of a Security service specification, **Principal** values have no standard format or interpretation, beyond serving to identify callers (and potential callers). This specification does not prescribe any usage of **Principal** values.

By representing **Principal** values as **sequence<octet>**, GIOP guarantees that ORBs may use domain-specific principal identification schemes; such values undergo no translation or interpretation during transmission. This allows bridges to translate or interpret these identifiers as needed when forwarding requests between different security domains.

15.3.5.4 Context

Context pseudo objects are encoded as **sequence<string>**. The strings occur in pairs. The first string in each pair is the context property name, and the second string in each pair is the associated value. If an operation has an IDL context clause but the client does not supply any properties matching the context clause at run time, an empty sequence is marshaled.

15.3.5.5 Exception

Exceptions are encoded as a string followed by exception members, if any. The string contains the RepositoryId for the exception, as defined in the Interface Repository chapter. Exception members (if any) are encoded in the same manner as a struct.

If an ORB receives a non-standard system exception that it does not support, or a user exception that is not defined as part of the operation's definition, the exception shall be mapped to UNKNOWN, with standard minor code set to 2 for a system exception, or set to 1 for a user exception.

15.3.6 Object References

Object references are encoded in OMG IDL (as described in Section 13.5, “Object Addressing,” on page 13-11). IOR profiles contain transport-specific addressing information, so there is no general-purpose IOR profile format defined for GIOP. Instead, this specification describes the general information model for GIOP profiles and provides a specific format for the IIOP (see “IIOP IOR Profiles” on page 15-52).

In general, GIOP profiles include at least these three elements:

1. The version number of the transport-specific protocol specification that the server supports.
2. The address of an endpoint for the transport protocol being used.
3. An opaque datum (an **object_key**, in the form of an octet sequence) used exclusively by the agent at the specified endpoint address to identify the object.

15.3.7 Abstract Interfaces

Abstract interfaces are encoded as a union with a **boolean** discriminator. The **union** has an *object reference* (see Section 15.3.6, “Object References,” on page 15-30) if the discriminator is **TRUE**, and a *value type* (see Section 15.3.4, “Value Types,” on page 15-15) if the discriminator is **FALSE**. The encoding of value types marshaled as abstract interfaces always includes **RepositoryId** information. If there is no indication whether a nil abstract interface represents a nil object reference or a null valuetype, it shall be encoded as a null valuetype.

15.4 GIOP Message Formats

GIOP has restriction on client and server roles with respect to initiating and receiving messages. For the purpose of GIOP versions 1.0 and 1.1, a client is the agent that opens a connection (see more details in Section 15.5.1, “Connection Management,” on page 15-47) and originates requests. Likewise, for GIOP versions 1.0 and 1.1, a server is an agent that accepts connections and receives requests.When Bidirectional GIOP is in use for GIOP protocol version 1.2 and 1.3, either side may originate messages, as specified in Section 15.8, “Bi-Directional GIOP,” on page 15-56.

GIOP message types are summarized in Table 15-3, which lists the message type names, whether the message is originated by client, server, or both, and the value used to identify the message type in GIOP message headers.

Table 15-3 GIOP Message Types and Originators

Message Type	**Originator**	**Value**	**GIOP Versions**
Request	Client	0	1.0, 1.1, 1.2, 1.3
Request	Both	0	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
Reply	Server	1	1.0, 1.1, 1.2, 1.3
Reply	Both	1	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
CancelRequest	Client	2	1.0, 1.1, 1.2, 1.3
CancelRequest	Both	2	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
LocateRequest	Client	3	1.0, 1.1, 1.2, 1.3
LocateRequest	Both	3	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
LocateReply	Server	4	1.0, 1.1, 1.2, 1.3
LocateReply	Both	4	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
CloseConnection	Server	5	1.0, 1.1, 1.2, 1.3
CloseConnection	Both	5	1.2, 1.3
MessageError	Both	6	1.0, 1.1, 1.2, 1.3
Fragment	Both	7	1.1, 1.2, 1.3

15.4.1 GIOP Message Header

All GIOP messages begin with the following header, defined in OMG IDL:

```
module GIOP { // IDL extended for version 1.1, 1.2, and 1.3
    struct Version {
        octet       major;
        octet       minor;
    };

#ifndef GIOP_1_1
    // GIOP 1.0
    enum MsgType_1_0 { // Renamed from MsgType
```

```
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError
    };

#else
    // GIOP 1.1
    enum MsgType_1_1 {
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError,
        Fragment            // GIOP 1.1 addition
    };
#endif  // GIOP_1_1

    // GIOP 1.0
    struct MessageHeader_1_0 { // Renamed from MessageHeader
        char                magic [4];
        Version             GIOP_version;
        boolean             byte_order;
        octet               message_type;
        unsigned long       message_size;
    };

    // GIOP 1.1
    struct MessageHeader_1_1 {
        char                magic [4];
        Version             GIOP_version;
        octet               flags;          // GIOP 1.1 change
        octet               message_type;
        unsigned long       message_size;
    };

    // GIOP 1.2, 1.3
    typedef MessageHeader_1_1 MessageHeader_1_2;
    typedef MessageHeader_1_1 MessageHeader_1_3;
};
```

The message header clearly identifies GIOP messages and their byte-ordering. The header is independent of byte ordering except for the field encoding message size.

- **magic** identifies GIOP messages. The value of this member is always the four (upper case) characters “GIOP,” encoded in ISO Latin-1 (8859.1).
- **GIOP_version** contains the version number of the GIOP protocol being used in the message. The version number applies to the transport-independent elements of this specification (i.e., the CDR and message formats) that constitute the GIOP. This is not equivalent to the IIOP version number (as described in Section 15.3.6, “Object References,” on page 15-30) though it has the same structure. The major GIOP version number of this specification is one (1); the minor versions are zero (0), one (1), and two (2).

A server implementation supporting a minor GIOP protocol version 1.n (with n > 0 and n < 3), must also be able to process GIOP messages having minor protocol version 1.m, with m less than n. A GIOP server, which receives a request having a greater minor version number than it supports, should respond with an error message having the highest minor version number that that server supports, and then close the connection.

A client should not send a GIOP message having a higher minor version number than that published by the server in the tag Internet IIOP Profile body of an IOR.

- **byte_order** (in GIOP 1.0 only) indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of **FALSE** (0) indicates big-endian byte ordering, and **TRUE** (1) indicates little-endian byte ordering.
- flags (in GIOP 1.1, 1.2, and 1.3) is an 8-bit octet. The least significant bit indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of **FALSE** (0) indicates big-endian byte ordering, and **TRUE** (1) indicates little-endian byte ordering. The byte order for fragment messages must match the byte order of the initial message that the fragment extends.

  The second least significant bit indicates whether or not more framents follow. A value of **FALSE** (0) indicates this message is the last fragment, and **TRUE** (1) indicates more fragments follow this message.

  The most significant 6 bits are reserved. These 6 bits must have value 0 for GIOP version 1.1, 1.2, and 1.3.
- **message_type** indicates the type of the message, according to Table 15-3; these correspond to enum values of type **MsgType**.
- **message_size** contains the number of octets in the message following the message header, encoded using the byte order specified in the byte order bit (the least significant bit) in the **flags** field (or using the byte_order field in GIOP 1.0). It refers to the size of the message body, not including the 12-byte message header. This count includes any alignment gaps and must match the size of the actual request parameters (plus any final padding bytes that may follow the parameters to have a fragment message terminate on an 8-byte boundary).

  A MARSHAL exception with minor code 7 indicates that fewer bytes were present in a message than indicated by the count. (This condition can arise if the sender sends a message in fragments, and the receiver detects that the final fragment was received but contained insufficient data for all parameters to be unmarshaled.).

  A MARSHAL exception with minor code 8 indicates that more bytes were present in a message than indicated by the count. Depending on the ORB implementation, this condition may be reported for the current message or the next message that is processed (when the receiver detects that the previous message is not immediately followed by the GIOP magic number).

The use of a message size of 0 with a **Request**, **LocateRequest**, **Reply**, or **LocateReply** message is reserved for future use.

For GIOP version 1.2, and 1.3, if the second least significant bit of **Flags** is 1, the sum of the **message_size** value and 12 must be evenly divisible by 8.

Messages with different GIOP minor versions may be mixed on the same underlying transport connection.

15.4.2 Request Message

Request messages encode CORBA object invocations, including attribute accessor operations, and **CORBA::Object** operations **get_interface** and **get_implementation**. Requests flow from client to server.

Request messages have three elements, encoded in this order:

- A GIOP message header
- A Request Header
- The Request Body

15.4.2.1 Request Header

The request header is specified as follows:

```
module GIOP { // IDL extended for version 1.1, 1.2, and 1.3

    // GIOP 1.0
    struct RequestHeader_1_0 { // Renamed from RequestHeader
        IOP::ServiceContextList       service_context;
        unsigned long                 request_id;
        boolean                       response_expected;
        sequence <octet>              object_key;
        string                        operation;
        CORBA::OctetSeq               requesting_principal;
    };

    // GIOP 1.1
    struct RequestHeader_1_1 {
        IOP::ServiceContextList       service_context;
        unsigned long                 request_id;
        boolean                       response_expected;
        octet                         reserved[3]; // Added in GIOP 1.1
        sequence <octet>              object_key;
        string                        operation;
        CORBA::OctetSeq               requesting_principal;
    };

    // GIOP 1.2, 1.3
```

```
    typedef short                   AddressingDisposition;
    const short                     KeyAddr = 0;
    const short                     ProfileAddr = 1;
    const short                     ReferenceAddr = 2;

    struct IORAddressingInfo {
        unsigned long                   selected_profile_index;
        IOP::IOR                        ior;
    };

    union TargetAddress switch (AddressingDisposition) {
        case KeyAddr:               sequence <octet> object_key;
        case ProfileAddr:           IOP::TaggedProfile profile;
        case ReferenceAddr:         IORAddressingInfo ior;
    };

    struct RequestHeader_1_2 {
        unsigned long                   request_id;
        octet                           response_flags;
        octet                           reserved[3];
        TargetAddress                   target;
        string                          operation;
        IOP::ServiceContextList         service_context;
        // requesting_principal not in GIOP 1.2 and 1.3
    };
    typedef RequestHeader_1_2 RequestHeader_1_3;
};
```

The members have the following definitions:

- **request_id** is used to associate reply messages with request messages (including **LocateRequest** messages). The client (requester) is responsible for generating values so that ambiguity is eliminated; specifically, a client must not re-use **request_id** values during a connection if: *(a)* the previous request containing that ID is still pending, or *(b)* if the previous request containing that ID was canceled and no reply was received. (See the semantics of the Section 15.4.4, "CancelRequest Message," on page 15-41).
- **response_flags** is set to 0x0 for a **SyncScope** of **NONE** and **WITH_TRANSPORT**. The flag is set to 0x1 for a **SyncScope** of **WITH_SERVER**. A non exception reply to a request message containing a **response_flags** value of 0x1 should contain an empty body, i.e. the equivalent of a void operation with no out/inout parameters. The flag is set to 0x3 for a **SyncScope** of **WITH_TARGET**. These values ensure interworking compatibility between this and previous versions of **GIOP**.

  For GIOP 1.0 and 1.1 a **response_expected** value of **TRUE** is treated like a **response_flags** value of \x03, and a **response_expected** value of **FALSE** is treated like a **response_flags** value of \x00.
- **reserved** is always set to **0** in GIOP 1.1. These three octets are reserved for future use.

- For GIOP 1.0 and 1.1, **object_key** identifies the object that is the target of the invocation. It is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.
- For GIOP 1.2, 1.3, **target** identifies the object that is the target of the invocation. The possible values of the union are:
  - **KeyAddr** is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.
  - **ProfileAddr** is the transport-specific GIOP profile selected for the target's IOR by the client ORB.
  - **IORAddressingInfo** is the full IOR of the target object. The **selected_profile_index** indicates the transport-specific GIOP profile that was selected by the client ORB. The first profile has an index of zero.
- **operation** is the IDL identifier naming, within the context of the interface (not a fully qualified scoped name), the operation being invoked. In the case of attribute accessors, the names are **_get_<attribute>** and **_set_<attribute>**. The case of the operation or attribute name must match the case of the operation name specified in the OMG IDL source for the interface being used.

  In the case of **CORBA::Object** operations that are defined in the CORBA Core (Section 4.3, "Object Reference Operations," on page 4-12) and that correspond to GIOP request messages, the operation names are **_interface**, **_is_a**, **_non_existent**, **_domain_managers,** and **_component**.

Note – The name **_get_domain_managers** is not used, to avoid conflict with a get operation invoked on a user defined attribute with name **domain_managers**.

  For GIOP 1.2 and later versions, only the operation name **_non_existent** shall be used.

  The correct operation name to use for GIOP 1.0 and 1.1 is **_non_existent**. Due to a typographical error in CORBA 2.0, 2.1, and 2.2, some legacy implementations of GIOP 1.0 and 1.1 respond to the operation name **_not_existent**. For maximum interoperability with such legacy implementations, new implementations of GIOP 1.0 and 1.1 may wish to respond to both operation names, **_non_existent** and **_not_existent**.

- **service_context** contains ORB service data being passed from the client to the server, encoded as described in Section 13.7, "Service Context," on page 13-28.
- **requesting_principal** contains a value identifying the requesting principal. It is provided to support the **BOA::get_principal** operation. The usage of the **requesting_principal** field is deprecated for GIOP versions 1.0 and 1.1. The field is not present in the request header for GIOP version 1.2, 1.3.

There is no padding after the request header when an unfragmented request message body is empty.

15.4.2.2 Request Body

In GIOP versions 1.0 and 1.1, request bodies are marshaled into the CDR encapsulation of the containing Message immediately following the Request Header. In GIOP version 1.2 and 1.3, the Request Body is always aligned on an 8-octet boundary. The fact that GIOP specifies the maximum alignment for any primitive type is 8 guarantees that the Request Body will not require remarshaling if the Message or Request header are modified. The data for the request body includes 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. 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.

For example, the request body for the following OMG IDL operation

```
double example (in short m, out string str, inout long p);
```

would be equivalent to this structure:

```
struct example_body {
    short           m;      // leftmost in or inout parameter
    long            p;      // ... to the rightmost
};
```

15.4.3 Reply Message

Reply messages are sent in response to **Request** messages if and only if the response expected flag in the request is set to **TRUE**. Replies include inout and out parameters, operation results, and may include exception values. In addition, Reply messages may provide object location information. In GIOP versions 1.0 and 1.1, replies flow only from server to client.

Reply messages have three elements, encoded in this order:

- A GIOP message header
- A *ReplyHeader structure*
- The reply body

15.4.3.1 Reply Header

The reply header is defined as follows:

```
module GIOP {                       // IDL extended for 1.2 and 1.3

#ifndef GIOP_1_2
```

```
    // GIOP 1.0 and 1.1
    enum ReplyStatusType_1_0 { // Renamed from ReplyStatusType
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    // GIOP 1.0
    struct ReplyHeader_1_0 { // Renamed from ReplyHeader
        IOP::ServiceContextList        service_context;
        unsigned long                  request_id;
        ReplyStatusType_1_0            reply_status;
    };

    // GIOP 1.1
    typedef ReplyHeader_1_0 ReplyHeader_1_1;
    // Same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2, 1.3
    enum ReplyStatusType_1_2 {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD,
        LOCATION_FORWARD_PERM,// new value for 1.2
        NEEDS_ADDRESSING_MODE // new value for 1.2
    };

    struct ReplyHeader_1_2 {
        unsigned long                  request_id;
        ReplyStatusType_1_2            reply_status;
        IOP::ServiceContextList        service_context;
    };
    typedef ReplyHeader_1_2 ReplyHeader_1_3;
#endif // GIOP_1_2
};
```

The members have the following definitions:

- **request_id** is used to associate replies with requests. It contains the same **request_id** value as the corresponding request.
- **reply_status** indicates the completion status of the associated request, and also determines part of the reply body contents. If no exception occurred and the operation completed successfully, the value is NO_EXCEPTION and the body contains return values. Otherwise the body
  - contains an exception, or
  - directs the client to reissue the request to an object at some other location, or
  - directs the client to supply more addressing information.

- **service_context** contains ORB service data being passed from the server to the client, encoded as described in Section 15.2.3, “GIOP Message Transfer,” on page 15-4.

There is no padding after the reply header when an unfragmented reply message body is empty.

15.4.3.2 Reply Body

In GIOP version 1.0 and 1.1, reply bodies are marshaled into the CDR encapsulation of the containing Message immediately following the Reply Header. In GIOP version 1.2 and 1.3, the Reply Body is always aligned on an 8-octet boundary. The fact that GIOP specifies the maximum alignment for any primitive type is 8 guarantees that the ReplyBody will not require remarshaling if the Message or the Reply Header are modified. The data for the reply body is determined by the value of **reply_status**. There are the following types of reply body:

- If the **reply_status** value is NO_EXCEPTION, the body is encoded as if it were a structure holding first any operation return value, then any **inout** and **out** parameters in the order in which they appear in the operation’s OMG IDL definition, from left to right. (That structure could be empty.)
- If the **reply_status** value is USER_EXCEPTION or SYSTEM_EXCEPTION, then the body contains the exception that was raised by the operation, encoded as described in Section 15.3.5.5, “Exception,” on page 15-30. (Only the user-defined exceptions listed in the operation’s OMG IDL definition may be raised.)

  When a GIOP Reply message contains a `**reply_status**' value of SYSTEM_EXCEPTION, the body of the Reply message conforms to the following structure:

```
module GIOP {                                        // IDL
      struct SystemExceptionReplyBody {
            string                     exception_id;
             unsigned long             minor_code_value;
            unsigned long              completion_status;
            };
};
```

  The high-order 20 bits of **minor_code_value** contain a 20-bit “Vendor Minor Codeset ID” (**VMCID**); the low-order 12 bits contain a minor code. A vendor (or group of vendors) wishing to define a specific set of system exception minor codes should obtain a unique **VMCID** from the OMG, and then use those 4096 minor codes as they see fit; for example, defining up to 4096 minor codes for each system exception. Any vendor may use the special **VMCID** of zero (0) without previous reservation, but minor code assignments in this codeset may conflict with other vendor's assignments, and use of the zero **VMCID** is officially deprecated.

Note – OMG standard minor codes are identified with the 20 bit **VMCID \x4f4d0**. This appears as the characters 'O' followed by the character 'M' on the wire, which is defined as a 32-bit constant called **OMGVMCID \x4f4d0000** (see Section 4.12.4, "Standard Minor Exception Codes," on page 4-72) so that allocated minor code numbers can be or-ed with it to obtain the **minor_code_value**.

- If the **reply_status** value is **LOCATION_FORWARD**, then the body contains an object reference (IOR) encoded as described in Section 15.3.6, "Object References," on page 15-30. The client ORB is responsible for re-sending the original request to that (different) object. This resending is transparent to the client program making the request.
- The usage of the **reply_status** value **LOCATION_FORWARD_PERM** behaves like the usage of **LOCATION_FORWARD**, but when used by a server it also provides an indication to the client that it may replace the old IOR with the new IOR. Both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.
- If the **reply_status** value is **NEEDS_ADDRESSING_MODE** then the body contains a **GIOP::AddressingDisposition**. The client ORB is responsible for re-sending the original request using the requested addressing mode. The resending is transparent to the client program making the request.

Note – Usage of LOCATATION_FORWARD_PERM is now deprecated, due to problems it causes with the semantics of the Object::hash() operation. LOCATATION_FORWARD_PERM features could be removed from some future GIOP versions if solutions to these problems are not provided.

For example, the reply body for a successful response (the value of **reply_status** is **NO_EXCEPTION**) to the *Request* example shown on page 15-37 would be equivalent to the following structure:

```
struct example_reply {
    double      return_value;     // return value
    string      str;
    long        p;                // ... to the rightmost
};
```

Note that the **object_key** field in any specific GIOP profile is server-relative, not absolute. Specifically, when a new object reference is received in a **LOCATION_FORWARD Reply** or in a **LocateReply** message, the **object_key** field embedded in the new object reference's GIOP profile may not have the same value as the **object_key** in the GIOP profile of the original object reference. For details on location forwarding, see Section 15.6, "Object Location," on page 15-49.

15.4.4 CancelRequest Message

CancelRequest messages may be sent, in GIOP versions 1.0 and 1.1, only from clients to servers. **CancelRequest** messages notify a server that the client is no longer expecting a reply for a specified pending **Request** or **LocateRequest** message.

CancelRequest messages have two elements, encoded in this order:

- A GIOP message header
- A **CancelRequestHeader**

15.4.4.1 Cancel Request Header

The cancel request header is defined as follows:

```
module GIOP {                                   // IDL
    struct CancelRequestHeader {
        unsigned long           request_id;
    };
};
```

The **request_id** member identifies the **Request** or **LocateRequest** message to which the cancel applies. This value is the same as the **request_id** value specified in the original **Request** or **LocateRequest** message.

When a client issues a cancel request message, it serves in an advisory capacity only. The server is not required to acknowledge the cancellation, and may subsequently send the corresponding reply. The client should have no expectation about whether a reply (including an exceptional one) arrives.

15.4.5 LocateRequest Message

LocateRequest messages may be sent from a client to a server to determine the following regarding a specified object reference:

- whether the current server is capable of directly receiving requests for the object reference, and if not,
- to what address requests for the object reference should be sent.

Note that this information is also provided through the **Request** message, but that some clients might prefer not to support retransmission of potentially large messages that might be implied by a **LOCATION_FORWARD** status in a **Reply** message. That is, client use of this represents a potential optimization.

LocateRequest messages have two elements, encoded in this order:

- A GIOP message header
- A **LocateRequestHeader**

15.4.5.1 *LocateRequest Header.*

The **LocateRequest** header is defined as follows:

```
module GIOP {                       // IDL extended for version 1.2 and 1.3

// GIOP 1.0
    struct LocateRequestHeader_1_0 {
                                    // Renamed LocationRequestHeader
        unsigned long           request_id;
        sequence <octet>        object_key;
    };

// GIOP 1.1
    typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;
    // Same Header contents for 1.0 and 1.1

// GIOP 1.2, 1.3
    struct LocateRequestHeader_1_2 {
        unsigned long           request_id;
        TargetAddress           target;
    };
    typedef LocateRequestHeader_1_2 LocateRequestHeader_1_3;
};
```

The members are defined as follows:

- **request_id** is used to associate **LocateReply** messages with **LocateRequest** ones. The client (requester) is responsible for generating values; see Section 15.4.2, "Request Message," on page 15-34 for the applicable rules.
- For GIOP 1.0 and 1.1, **object_key** identifies the object being located. In an IIOP context, this value is obtained from the **object_key** field from the encapsulated **IIOP::ProfileBody** in the IIOP profile of the IOR for the target object. When GIOP is mapped to other transports, their IOR profiles must also contain an appropriate corresponding value. This value is only meaningful to the server and is not interpreted or modified by the client.
- For GIOP 1.2, 1.3, target identifies the object being located. The possible values of this union are:
  - **KeyAddr** is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.
  - **ProfileAddr** is the transport-specific GIOP profile selected for the target's IOR by the client ORB.
  - **IORAddressingInfo** is the full IOR of the target object. The **selected_profile_index** indicates the transport-specific GIOP profile that was selected by the client ORB.

See Section 15.6, "Object Location," on page 15-49 for details on the use of **LocateRequest**.

15.4.6 LocateReply Message

LocateReply messages are sent from servers to clients in response to **LocateRequest** messages. In GIOP versions 1.0 and 1.1 the **LocateReply** message is only sent from the server to the client.

A **LocateReply** message has three elements, encoded in this order:

1. A GIOP message header
2. A **LocateReplyHeader**
3. The locate reply body

15.4.6.1 Locate Reply Header

The locate reply header is defined as follows:

```
module GIOP {                           // IDL extended for GIOP 1.2 and 1.3
#ifndef GIOP_1_2
    // GIOP 1.0 and 1.1
    enum LocateStatusType_1_0 {// Renamed from LocateStatusType
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

    // GIOP 1.0
    struct LocateReplyHeader_1_0 {// Renamed from LocateReplyHeader
        unsigned long           request_id;
        LocateStatusType_1_0    locate_status;
    };

    // GIOP 1.1
    typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;
    // same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2, 1.3
    enum LocateStatusType_1_2 {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD,
        OBJECT_FORWARD_PERM,            // new value for GIOP 1.2
        LOC_SYSTEM_EXCEPTION,           // new value for GIOP 1.2
        LOC_NEEDS_ADDRESSING_MODE       // new value for GIOP 1.2
    };

    struct LocateReplyHeader_1_2 {
        unsigned long           request_id;
        LocateStatusType_1_2    locate_status;
```

```
    };
    typedef LocateReplyHeader_1_2 LocateReplyHeader_1_3;
#endif // GIOP_1_2
};
```

The members have the following definitions:

- **request_id** - is used to associate replies with requests. This member contains the same **request_id** value as the corresponding **LocateRequest** message.
- **locate_status -** the value of this member is used to determine whether a **LocateReply** body exists. Values are:
  - **UNKNOWN_OBJECT -** the object specified in the corresponding **LocateRequest** message is unknown to the server; no body exists.
  - **OBJECT_HERE -** this server (the originator of the **LocateReply** message) can directly receive requests for the specified object; no body exists.
  - **OBJECT_FORWARD** and **OBJECT_FORWARD_PERM -** a **LocateReply** body exists.
  - **LOC_SYSTEM_EXCEPTION** - a **LocateReply** body exists.
  - **LOC_NEEDS_ADDRESSING_MODE** - a **LocateReply** body exists.

15.4.6.2 *LocateReply Body*

The body is empty, except for the following cases:

- If the **LocateStatus** value is **OBJECT_FORWARD** or **OBJECT_FORWARD_PERM**, the body contains an object reference (IOR) that may be used as the target for requests to the object specified in the **LocateRequest** message. The usage of **OBJECT_FORWARD_PERM** behaves like the usage of **OBJECT_FORWARD**, but when used by the server it also provides an indication to the client that it may replace the old IOR with the new IOR. When using **OBJECT_FORWARD_PERM**, both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.
- If the **LocateStatus** value is **LOC_SYSTEM_EXCEPTION**, the body contains a marshaled **GIOP::SystemExceptionReplyBody**.
- If the **LocateStatus** value is **LOC_NEEDS_ADDRESSING_MODE,** then the body contains a **GIOP::AddressingDisposition**. The client ORB is responsible for re-sending the **LocateRequest** using the requested addressing mode.

Note – Usage of **OBJECT_FORWARD_PERM** is now deprecated, due to problems it causes with the semantics of the **Object::hash** operation. **OBJECT_FORWARD_PERM** features could be removed from some future GIOP versions if solutions to these problems are not provided.

LocateReply bodies are marshaled immediately following the LocateReply header.

15.4.6.3 Handling ForwardRequest Exception from ServantLocator

If the **ServantLocator** in a POA raises a ForwardRequest exception the ORB shall send a **LocateReply** message to the client with **locate_status** set to **OBJECT_FORWARD**, and with the body containing the object reference from the ForwardRequest exception's **forward_reference** field.

15.4.7 CloseConnection Message

CloseConnection messages are sent only by servers in GIOP protocol versions 1.0 and 1.1. They inform clients that the server intends to close the connection and must not be expected to provide further responses. Moreover, clients know that any requests for which they are awaiting replies will never be processed, and may safely be reissued (on another connection). In GIOP version 1.2 and 1.3 both sides of the connection may send the **CloseConnection** message.

The **CloseConnection** message consists only of the GIOP message header, identifying the message type.

For details on the usage of **CloseConnection** messages, see Section 15.5.1, "Connection Management," on page 15-47.

15.4.8 MessageError Message

The **MessageError** message is sent in response to any GIOP message whose version number or message type is unknown to the recipient or any message received whose header is not properly formed (e.g., has the wrong magic value). Error handling is context-specific.

The **MessageError** message consists only of the GIOP message header, identifying the message type.

15.4.9 Fragment Message

This message is added in GIOP 1.1.

The **Fragment** message is sent following a previous request or response message that has the more fragments bit set to **TRUE** in the **flags** field.

All of the GIOP messages begin with a GIOP header. One of the fields of this header is the **message_size** field, a 32-bit unsigned number giving the number of bytes in the message following the header. Unfortunately, when actually constructing a GIOP **Request** or **Reply** message, it is sometimes impractical or undesirable to ascertain the total size of the message at the stage of message construction where the message header has to be written. GIOP 1.1 provides an alternative indication of the size of the message, for use in those cases.

In GIOP 1.1, a **Request** or **Reply** message can be broken into multiple fragments. In GIOP 1.2 and 1.3, a **Request**, **Reply**, **LocateRequest**, or **LocateReply** message can be broken into multiple fragment. The first fragment is a regular message (e.g.,

Request or **Reply**) with the **more** fragments bit in the **flags** field set to **TRUE**. This initial fragment can be followed by one or more messages using the fragment messages. The last fragment shall have the more fragment bit in the flag field set to **FALSE**.

A **CancelRequest** message may be sent by the client before the final fragment of the message being sent. In this case, the server should assume no more fragments will follow.

Note – A GIOP client that fragments the header of a **Request** message before sending the request ID may not send a **CancelRequest** message pertaining to that request ID and may not send another **Request** message until after the request ID is sent.

A primitive data type of 8 bytes or smaller should never be broken across two fragments.

In GIOP 1.1, the data in a fragment is marshaled with alignment relative to its position in the fragment, not relative to its position in the whole unfragmented message.

For GIOP version 1.2 and 1.3, the total length (including the message header) of a fragment other than the final fragment of a fragmented message are required to be a multiple of 8 bytes in length, allowing bridges to defragment and/or refragment messages without having to remarshal the encoded data to insert or remove padding.

For GIOP version 1.2 and 1.3, a fragment header is included in the message, immediately after the GIOP message header and before the fragment data. The request ID, in the fragment header, has the same value as that used in the original message associated with the fragment.

The byte order and GIOP protocol version of a fragment shall be the same as that of the message it continues.

```
module GIOP {// IDL extension for GIOP 1.2 and 1.3
    // GIOP 1.2, 1.3
    struct FragmentHeader_1_2 {
        unsigned long request_id;
    };
    typedef FragmentHeader_1_2 FragmentHeader_1_3;
};
```

15.5 GIOP Message Transport

The GIOP is designed to be implementable on a wide range of transport protocols. The GIOP definition makes the following assumptions regarding transport behavior:

- The transport is connection-oriented. GIOP uses connections to define the scope and extent of request IDs.
- The transport is reliable. Specifically, the transport guarantees that bytes are delivered in the order they are sent, at most once, and that some positive acknowledgment of delivery is available.

- The transport can be viewed as a byte stream. No arbitrary message size limitations, fragmentation, or alignments are enforced.
- The transport provides some reasonable notification of disorderly connection loss. If the peer process aborts, the peer host crashes, or network connectivity is lost, a connection owner should receive some notification of this condition.
- The transport's model for initiating connections can be mapped onto the general connection model of TCP/IP. Specifically, an agent (described herein as a server) publishes a known network address in an IOR, which is used by the client when initiating a connection.

The server does not actively initiate connections, but is prepared to accept requests to connect (i.e., it *listens* for connections in TCP/IP terms). Another agent that knows the address (called a client) can attempt to initiate connections by sending *connect* requests to the address. The listening server may *accept* the request, forming a new, unique connection with the client, or it may *reject* the request (e.g., due to lack of resources). Once a connection is open, either side may *close* the connection. (See Section 15.5.1, "Connection Management," on page 15-47 for semantic issues related to connection closure.) A candidate transport might not directly support this specific connection model; it is only necessary that the transport's model can be mapped onto this view.

15.5.1 Connection Management

For the purposes of this discussion, the roles client and server are defined as follows:

- A client initiates the connection, presumably using addressing information found in an object reference (IOR) for an object to which it intends to send requests.
- A server accepts connections, but does not initiate them.

These terms only denote roles with respect to a connection. They do not have any implications for ORB or application architectures.

In GIOP protocol versions 1.0 and 1.1, connections are not symmetrical. Only clients can send **Request**, **LocateRequest**, and **CancelRequest** messages over a connection, in GIOP 1.0 and 1.1. In all GIOP versions, a server can send **Reply**, **LocateReply**, and **CloseConnection** messages over a connection; however, in GIOP 1.2, 1.3 the client can send them as well. Either client or server can send **MessageError** messages, in GIOP 1.0 and 1.1.

If multiple GIOP versions are used on an underlying transport connection, the highest GIOP version used on the connection can be used for handling the close. A **CloseConnection** message sent using any GIOP version applies to all GIOP versions used on the connection (i.e., the underlying transport connection is closed for all GIOP versions). In particular, if GIOP version 1.2 or higher has been used on the connection, the client can send the **CloseConnection** message by using the highest GIOP version in use.

Only GIOP messages are sent over GIOP connections.

Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection. Request IDs may be re-used if there is no possibility that the previous request using the ID may still have a pending reply. Note that cancellation does not guarantee no reply will be sent. It is the responsibility of the client to generate and assign request IDs. Request IDs must be unique among both **Request** and **LocateRequest** messages.

15.5.1.1 Connection Closure

Connections can be closed in two ways: orderly shutdown, or abortive disconnect.

For GIOP versions 1.0, and 1.1:

- Orderly shutdown is initiated by servers sending a **CloseConnection** message, or by clients just closing down a connection.
- Orderly shutdown may be initiated by the client at any time.
- A server may not initiate shutdown if it has begun processing any requests for which it has not either received a **CancelRequest** or sent a corresponding reply.
- If a client detects connection closure without receiving a **CloseConnection** message, it must assume an abortive disconnect has occurred, and treat the condition as an error.

For GIOP Version 1.2, 1.3:

- Orderly shutdown is initiated by either the originating client ORB (connection initiator) or by the server ORB (connection responder) sending a **CloseConnection** message
- If the ORB sending the **CloseConnection** is a server, or bidirectional GIOP is in use, the sending ORB must not currently be processing any Requests from the other side.
- The ORB that sends the **CloseConnection** must not send any messages after the **CloseConnection**.
- If either ORB detects connection closure without receiving a **CloseConnection** message, it must assume an abortive disconnect has occurred, and treat the condition as an error.
- If bidirectional GIOP is in use, the conditions of Section 15.8, “Bi-Directional GIOP,” on page 15-56 apply.

For all uses of **CloseConnection** (for GIOP versions 1.0, 1.1, 1.2, and 1.3):

- If there are any pending non-oneway requests, which were initiated on a connection by the ORB shutting down that connection, the connection-peer ORB should consider them as canceled.
- If an ORB receives a **CloseConnection** message from its connection-peer ORB, it should assume that any outstanding messages (i.e., without replies) were received after the connection-peer ORB sent the CloseConnection message, were not processed, and may be safely re-sent on a new connection.

- After issuing a **CloseConnection** message, the issuing ORB may close the connection. Some transport protocols (not including TCP) do not provide an "orderly disconnect" capability, guaranteeing reliable delivery of the last message sent. When GIOP is used with such protocols, an additional handshake needs to be provided as part of the mapping to that protocol's connection mechanisms, to guarantee that both ends of the connection understand the disposition of any outstanding GIOP requests.

15.5.1.2 Multiplexing Connections

A client, if it chooses, may send requests to multiple target objects over the same connection, provided that the connection's server side is capable of responding to requests for the objects. It is the responsibility of the client to optimize resource usage by reusing connections, if it wishes. If not, the client may open a new connection for each active object supported by the server, although this behavior should be avoided.

15.5.2 Message Ordering

Only the client (connection originator) may send **Request, LocateRequest, and CancelRequest** messages, if Bi-Directional GIOP is not in use.

Clients may have multiple pending requests. A client need not wait for a reply from a previous request before sending another request.

Servers may reply to pending requests in any order. **Reply** messages are not required to be in the same order as the corresponding **Requests**.

The ordering restrictions regarding connection closure mentioned in Connection Management, above, are also noted here. Servers may only issue **CloseConnection** messages when **Reply** messages have been sent in response to all received **Request** messages that require replies.

15.6 Object Location

The GIOP 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*, *object server process*, *Inter-ORB bridge*, and so forth). It merely implies the existence of some agent with which a connection may be opened, and to which requests may be sent.

The "agent" (owner of the server side of a connection) 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 an Inter-ORB bridge that transforms the request and passes it on to another process or ORB. From GIOP's perspective, it is only important that requests 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 **Request** messages sent to the agent would result in either exceptions or replies with **LOCATION_FORWARD** status, providing new addresses to which requests may be sent. Such agents would also respond to **LocateRequest** messages with appropriate **LocateReply** 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 (perhaps during the same connection).

Agents are not required to implement location forwarding mechanisms. An agent can be implemented with the policy that a connection either supports direct access to an object, or returns exceptions. Such an ORB (or inter-ORB bridge) always return **LocateReply** messages with either **OBJECT_HERE** or **UNKNOWN_OBJECT** status, and never **OBJECT_FORWARD** status.

Clients must, however, be able to accept and process **Reply** messages with **LOCATION_FORWARD** status, since any ORB may choose to implement a location service. Whether a client chooses to send **LocateRequest** messages is at the discretion of the client. For example, if the client routinely expected to see **LOCATION_FORWARD** replies when using the address in an object reference, it might always send **LocateRequest** messages to objects for which it has no recorded forwarding address. If a client sends **LocateRequest** messages, it should be prepared to accept **LocateReply** messages.

A client shall not make any assumptions about the longevity of object addresses returned by **LOCATION_FORWARD** (**OBJECT_FORWARD**) mechanisms. Once a connection based on location-forwarding information is closed, a client can attempt to reuse the forwarding information it has, but, if that fails, it shall restart the location process using the original address specified in the initial object reference.

For GIOP version 1.2 and 1.3, the usage of **LOCATION_FORWARD_PERM** (**OBJECT_FORWARD_PERM**) behaves like the usage of **LOCATION_FORWARD** (**OBJECT_FORWARD**), but when used by the server it also provides an indication to the client that it may replace the old IOR with the new IOR. When using **LOCATION_FORWARD_PERM** (**OBJECT_FORWARD_PERM**), both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.

Note – Usage of **LOCATION_FORWARD_PERM** and **OBJECT_FORWARD_PERM** is now deprecated, due to problems it causes with the semantics of the **Object::hash** operation. **LOCATION_FORWARD_PERM** and **OBJECT_FORWARD_PERM** features could be removed from some future GIOP versions if solutions to these problems are not provided.

Even after performing successful invocations using an address, a client should be prepared to be forwarded. The only object address that a client should expect to continue working reliably is the one in the initial object reference. If an invocation using that address returns **UNKNOWN_OBJECT**, the object should be deemed non-existent.

In general, the implementation 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.

15.7 Internet Inter-ORB Protocol (IIOP)

The baseline transport specified for GIOP is TCP/IP[5]. Specific APIs for libraries supporting TCP/IP may vary, so this discussion is limited to an abstract view of TCP/IP and management of its connections. The mapping of GIOP message transfer to TCP/IP connections is called the Internet Inter-ORB Protocol (IIOP).

IIOP 1.0 is based on GIOP 1.0.

IIOP 1.1 can be based on either GIOP 1.0 or 1.1. An IIOP 1.1 client must support GIOP 1.1, and may also support GIOP 1.0. An IIOP 1.1 server must support processing both GIOP 1.0 and GIOP 1.1 messages.

IIOP 1.2 can be based on any of the GIOP minor versions 1.0, 1.1, or 1.2. An IIOP 1.2 client must support GIOP 1.2, and may also support lesser GIOP minor versions. An IIOP 1.2 server must also support processing messages with all lesser GIOP versions.

IIOP 1.3 can be based on any of the GIOP minor versions 1.0, 1.1, 1.2, or 1.3. An IIOP 1.3 client must support GIOP 1.3, and may also support lesser GIOP minor versions. An IIOP 1.3 server must also support processing messages with all lesser GIOP versions.

Conformance to IIOP versions 1.1, 1.2, and 1.3 requires support of Limited-Profile IOR conformance (see Section 13.6.2, “Interoperable Object References: IORs,” on page 13-14), specifically for the IIOP IOR Profile. As of CORBA 2.4, this limited IOR conformance is deprecated, and ORBs implementing IIOP are strongly recommended to support Full IOR conformance. Some future IIOP versions could require support of Full IOR conformance.

5. Postel, J., “Transmission Control Protocol – DARPA Internet Program Protocol Specification,” RFC-793, Information Sciences Institute, September 1981

15.7.1 TCP/IP Connection Usage

Agents that are capable of accepting object requests or providing locations for objects (i.e., servers) publish TCP/IP addresses in IORs, as described in Section 15.7.2, "IIOP IOR Profiles," on page 15-52. A TCP/IP address consists of an IP host address, typically represented by a host name, and a TCP port number. Servers must listen for connection requests.

A client needing an object's services must initiate a connection with the address specified in the IOR, with a connect request.

The listening server may accept or reject the connection. In general, servers should accept connection requests if possible, but ORBs are free to establish any desired policy for connection acceptance (e.g., to enforce fairness or optimize resource usage).

Once a connection is accepted, the client may send **Request, LocateRequest**, or **CancelRequest** messages by writing to the TCP/IP socket it owns for the connection. The server may send **Reply, LocateReply**, and **CloseConnection** messages by writing to its TCP/IP connection. In GIOP 1.2, and 1.3, the client may send the **CloseConnection** message, and if BiDirectional GIOP is in use, the client may also send **Reply** and **LocateReply** messages.

After receiving a **CloseConnection** message, an ORB must close the TCP/IP connection. After sending a **CloseConnection**, an ORB may close the TCP/IP connection immediately, or may delay closing the connection until it receives an indication that the other side has closed the connection. For maximum interoperability with ORBs using TCP implementations that do not properly implement orderly shutdown, an ORB may wish to only shutdown the sending side of the connection, and then read any incoming data until it receives an indication that the other side has also shutdown, at which point the TCP connection can be closed completely.

Given TCP/IP's flow control mechanism, it is possible to create deadlock situations between clients and servers if both sides of a connection send large amounts of data on a connection (or two different connections between the same processes) and do not read incoming data. Both processes may block on write operations, and never resume. It is the responsibility of both clients and servers to avoid creating deadlock by reading incoming messages and avoiding blocking when writing messages, by providing separate threads for reading and writing, or any other workable approach. ORBs are free to adopt any desired implementation strategy, but should provide robust behavior.

15.7.2 IIOP IOR Profiles

IIOP profiles, identifying individual objects accessible through the Internet Inter-ORB Protocol, have the following form:

```
module IIOP {   // IDL extended for version 1.1, 1.2, and 1.3
    struct Version {
        octet           major;
        octet           minor;
    };
```

```
    struct ProfileBody_1_0 {// renamed from ProfileBody
        Version             iiop_version;
        string              host;
        unsigned short      port;
        sequence <octet>    object_key;
    };

    struct ProfileBody_1_1 {// also used for 1.2 and 1.3
        Version             iiop_version;
        string              host;
        unsigned short      port;
        sequence <octet>    object_key;

    // Added in 1.1 unchanged for 1.2 and 1.3
        sequence <IOP::TaggedComponent> components;
    };
};
```

IIOP Profile version number:

- Indicates the IIOP protocol version.
- Major number can stay the same if the new changes are backward compatible.
- Clients with lower minor version can attempt to invoke objects with higher minor version number by using only the information defined in the lower minor version protocol (ignore the extra information).

Profiles supporting only IIOP version 1.0 use the **ProfileBody_1_0** structure, while those supporting IIOP version 1.1 or 1.2 or 1.3 use the **ProfileBody_1_1** structure. An instance of one of these structure types is marshaled into an encapsulation octet stream. This encapsulation (a **sequence <octet>**) becomes the **profile_data** member of the **IOP::TaggedProfile** structure representing the IIOP profile in an IOR, and the tag has the value **TAG_INTERNET_IOP** (as defined earlier).

The version number published in the Tag Internet IIOP Profile body signals the highest GIOP minor version number that the server supports at the time of publication of the IOR.

If the major revision number is 1, and the minor revision number is greater than 0, then the length of the encapsulated profile may exceed the total size of components defined in this specification for profiles with minor revision number 0. ORBs that support only revision 1.0 IIOP profiles must ignore any data in the profile that occurs after the **object_key**. If the revision of the profile is 1.0, there shall be no extra data in the profile (i.e., the length of the encapsulated profile must agree with the total size of components defined for version 1.0).

For Version 1.2 and 1.3 of IIOP, no order of use is prescribed in the case where more than one TAG Internet IOP Profile is present in an IOR.

The members of **IIOP::ProfileBody_1_0 and IOP::ProfileBody_1_1** are defined as follows:

- **iiop_version** describes the version of IIOP that the agent at the specified address is prepared to receive. When an agent generates IIOP profiles specifying a particular version, it must be able to accept messages complying with the specified version or any previous minor version (i.e., any smaller version number). The major version number of this specification is 1; the minor versions defined to date are 0, 1, and 2. Compliant ORBs must generate version 1.1 profiles, and must accept any profile with a major version of 1, regardless of the minor version number. If the minor version number is 0, the encapsulation is fully described by the **ProfileBody_1_0** structure. If the minor version number is 1 or 2, the encapsulation is fully described by the **ProfileBody_1_1** structure. If the minor version number is greater than 2, then the length of the encapsulated profile may exceed the total size of components defined in this specification for profiles with minor version number 1 or 2. ORBs that support only version 1.1 or 1.2 IIOP profiles must ignore, but preserve, any data in the profile that occurs after the **components** member, for IIOP profiles with minor version greater than 1.2.

Note – As of version 1.2 of GIOP and IIOP and minor versions beyond, the minor version in the **TAG_INTERNET_IOP** profile signals the highest minor revision of GIOP supported by the server at the time of publication of the IOR.

- **host** identifies the Internet host to which GIOP messages for the specified object may be sent. In order to promote a very large (Internet-wide) scope for the object reference, this will typically be the fully qualified domain name of the host, rather than an unqualified (or partially qualified) name. However, per Internet standards, the host string may also contain a host address expressed in standard "dotted decimal" form (e.g., "192.231.79.52").
- **port** contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.
- **object_key** is an opaque value supplied by the agent producing the IOR. This value will be used in request messages to identify the object to which the request is directed. An agent that generates an object key value must be able to map the value unambiguously onto the corresponding object when routing requests internally.
- **components** is a sequence of **TaggedComponent**, which contains additional information that may be used in making invocations on the object described by this profile. **TaggedComponent**s that apply to IIOP 1.2 are described below in Section 15.7.3, "IIOP IOR Profile Components," on page 15-55. Other components may be included to support enhanced versions of IIOP, to support ORB services such as security, and to support other GIOPs, ESIOPs, and proprietary protocols. If an implementation puts a non-standard component in an IOR, it cannot be assured that any or all non-standard components will remain in the IOR.

  The relationship between the IIOP protocol version and component support conformance requirements is as follows:

- Each IIOP version specifies a set of standard components and the conformance rules for that version. These rules specify which components are mandatory and which are optional. A conformant implementation has to conform to these rules, and is not required to conform to more than these rules.
- New components can be added, but they do not become part of the versions conformance rules.
- When there is a need to specify conformance rules that include the new components, there will be a need to create a new IIOP version.

Note that host addresses are restricted in this version of IIOP to be Class A, B, or C Internet addresses. That is, Class D (multi-cast) addresses are not allowed. Such addresses are reserved for use in future versions of IIOP.

Agents may freely choose TCP port numbers for communication; IIOP supports multiple agents per host.

15.7.3 IIOP IOR Profile Components

The following components are part of IIOP 1.1, 1.2, and 1.3 conformance. All these components are optional.

- **TAG_ORB_TYPE**
- **TAG_CODE_SETS**
- **TAG_SEC_NAME**
- **TAG_ASSOCIATION_OPTIONS**
- **TAG_GENERIC_SEC_MECH**
- **TAG_SSL_SEC_TRANS**
- **TAG_SPKM_1_SEC_MECH**
- **TAG_SPKM_2_SEC_MECH**
- **TAG_KerberosV5_SEC_MECH**
- **TAG_CSI_ECMA_Secret_SEC_MECH**
- **TAG_CSI_ECMA_Hybrid_SEC_MECH**
- **TAG_SSL_SEC_TRANS**
- **TAG_CSI_ECMA_Public_SEC_MECH**
- **TAG_FIREWALL_TRANS**
- **TAG_JAVA_CODEBASE**
- **TAG_TRANSACTION_POLICY**
- **TAG_MESSAGE_ROUTERS**
- **TAG_INET_SEC_TRANS**

The following components are part of IIOP 1.2, and 1.3 conformance. All these components are optional.

- **TAG_ALTERNATE_IIOP_ADDRESS**
- **TAG_POLICIES**
- **TAG_DCE_STRING_BINDING**
- **TAG_DCE_BINDING_NAME**
- **TAG_DCE_NO_PIPES**
- **TAG_DCE_MECH**
- **TAG_COMPLETE_OBJECT_KEY**
- **TAG_ENDPOINT_ID_POSITION**
- **TAG_LOCATION_POLICY**
- **TAG_OTS_POLICY**
- **TAG_INV_POLICY**
- **TAG_CSI_SEC_MECH_LIST**
- **TAG_NULL_TAG**
- **TAG_SECIOP_SEC_TRANS**
- **TAG_TLS_SEC_TRANS**
- **TAG_ACTIVITY_POLICY**

15.8 Bi-Directional GIOP

The specification of GIOP connection management, in GIOP minor versions 1.0 and 1.1, states that connections are not symmetrical. For example, only clients that initialize connections can send requests, and only servers that accept connections can receive them.

This GIOP 1.0 and 1.1 restriction gives rise to significant difficulties when operating across firewalls. It is common for firewalls not to allow incoming connections, except to certain well-known and carefully configured hosts, such as dedicated HTTP or FTP servers. For most CORBA-over-the-internet applications it is not practicable to require that all potential client firewalls install GIOP proxies to allow incoming connections, or that any entities receiving callbacks will require prior configuration of the firewall proxy.

An applet, for example, downloaded to a host inside such a firewall will be restricted in that it cannot receive requests from outside the firewall on any object it creates, as no host outside the firewall will be able to connect to the applet through the client's firewall, even though the applet in question would typically only expect callbacks from the server it initially registered with.

In order to circumvent this unnecessary restriction, GIOP minor protocol version 1.2 or 1.3 specifies that the asymmetry stipulation above be relaxed in cases where the client and the server agree on it. In these cases, the client (the applet in the above case) would still initiate the connection to the server, but any requests from the server on any objects.

The client creates an object for exporting to a server, and arranges that the server receive an IOR for the object. The most common use case would be for the client to pass the IOR as a parameter in a GIOP request, but other mechanisms are possible, such as the use of a Name Service. If the client ORB policy permits bi-directional use of a connection, a Request message should contain an **IOP::ServiceContext** structure in its Request header, which indicates that this GIOP connection is bi-directional. The service context may provide additional information that the server may need to invoke the callback object. To determine whether an ORB may support bi-directional GIOP new policies has been defined (Section 15.9, "Bi-directional GIOP policy," on page 15-60).

Each mapping of GIOP to a particular transport should define a transport-specific bi-directional service context, and have an **IOP::ServiceId** allocated by the OMG. It is recommended that names for this service context follows the pattern *BiDir<protocolname>*, where <protocol name> identifies a mapping of GIOP to a transport protocol (e.g., for IIOP the name is **BiDirIIOP**). The service context for bi-directional IIOP is defined in Section 15.8.1, "Bi-Directional IIOP," on page 15-58.

The server receives the Request, which contains a bi-directional **IOP::ServiceContext**. If the server supports bi-directional connections for that protocol, it may now send invocations along the same connection to any object that supports the particular protocol and matches the particular location information found in the bi-directional service context. If the server does not support bi-directional connections for that protocol, the service context can be ignored.

The data encapsulated in the **BiDirIIOPServiceContext** structure (see below), which is identified by the **ServiceId BI_DIR_IIOP** as defined in Section 13.7, "Service Context," on page 13-28, allows the ORB to determine whether it needs to open a new connection in order to invoke on an object. If a host and port pair in a listen_point list matches a host and port of an object to which it does not yet have a connection (a callback object newly received, for instance), rather than open a new connection, the server may re-use any of the connections on which the listen_point data was received.

A server talking to a client on a bi-directional GIOP connection can use any message type traditionally used by clients only, so it can use **Request**, **LocateRequest**, **CancelRequest**, **MessageError**, and **Fragment** (for a **Request** or **LocateRequest**). Similarly the client can use message types traditionally used only by servers: **Reply**, **LocateReply**, **MessageError**, **CloseConnection**, and **Fragment** (for a **Reply** or **LocateReply**).

CloseConnection messages are a special case however. Either ORB may send a **CloseConnection** message, but the conditions in Section 15.5.1, "Connection Management," on page 15-47 apply.

Bi-directional GIOP connections modify the behavior of Request IDs. In the GIOP specification, Section 15.5.1, “Connection Management,” on page 15-47, it is noted that “Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection.” This property of unambiguous association of requests and replies must be preserved while permitting each end to generate Request Ids for new requests independently. To ensure this, on a connection that is used bi-directionally in GIOP 1.2, and 1.3, the connection originator shall assign only even valued Request IDs and the other side of the connection shall assign only odd valued Request IDs. This requirement applies to the full lifetime of the connection, even before a **BiDirIIOPServiceContext** is transmitted. A connection on which this regime of Request ID assignment is not used, shall never be used to transmit bi-directional GIOP 1.2, or 1.3 messages.

It should be noted that a single-threaded ORB needs to perform event checking on the connection, in case a **Request** from the other endpoint arrives in the window between it sending its own **Request** and receiving the corresponding reply; otherwise a client and server could send **Request**s simultaneously, resulting in deadlock. If the client cannot support event checking, it must not indicate that bi-directionality is supported. If the server cannot support event checking, it must not make callbacks along the same connection even if the connection indicates it is supported.

A server making a callback to a client cannot specify its own bi-directional service context – only the client can announce the connection's bi-directionality.

An important security issue should be observed in the use of bi-directional GIOP. In the absence of other security mechanisms, a malicious client may claim that its connection is Bi-Directional for use with any host and port it chooses. In particular it may specify the host and port of security sensitive objects not even resident on its host. All the client has to do is pass the host and port in the listen data service context and the server may then invoke a masquerading object instead. In general, and in the absence of other security mechanisms, a server that has accepted an incoming connection has no way to discover the identity or verify the integrity of the client that initiated the connection. If the server has doubts in the integrity of the client, it is recommended that bi-directional GIOP is not used.

15.8.1 Bi-Directional IIOP

The **IOP::ServiceContext** used to support bi-directional IIOP contains a **BiDirIIOPServiceContext** structure as defined below:

```
// IDL
module IIOP {

    struct ListenPoint {
        string   host;
        unsigned short port;
    };

    typedef sequence<ListenPoint> ListenPointList;
```

```
    struct BiDirIIOPServiceContext {
        ListenPointList listen_points;
    };
};
```

The data encapsulated in the **BiDirIIOPServiceContext** structure, which is identified by the ServiceId **BI_DIR_IIOP** as defined in Section 13.7, "Service Context," on page 13-28, allows the ORB, which intends to open a new connection in order to invoke on an object, to look up its list of active client-initiated connections and examine the structures associated with them, if any. If a **host** and **port** pair in a **listen_points** list matches a host and port, which the ORB intends to open a connection to, rather than open a new connection to that **listen_point**, the server may re-use any of the connections that were initiated by the client on which the listen point data was received.

The **host** element of the structure should contain whatever values the client may use in the IORs it creates. The rules for **host** and **port** are identical to the rules for the IIOP IOR **ProfileBody_1_1 host** and **port** elements; see Section 15.7.2, "IIOP IOR Profiles," on page 15-52. Note that if the server wishes to make a callback connection to the client in the standard way, it must use the values from the client object's IOR, not the values from this **BiDirIIOPServiceContext** structure; these values are only to be used for bi-directional GIOP support.

The **BI_DIR_IIOP** service context may be sent by a client at any point in a connection's lifetime. The **listen_points** specified therein must supplement any **listen_points** already sent on the connection, rather than replacing the existing points.

If a client supports a secure connection mechanism, such as SECIOP or IIOP/SSL, and sends a **BI_DIR_IIOP** service context over an insecure connection, the **host** and **port** endpoints listed in the **BI_DIR_IIOP** should not contain the details of the secure connection mechanism if insecure callbacks to the client's secure objects would be a violation of the client's security policy.

It is the ORB's responsibility to ensure that an IOR contains an appropriate address.

15.8.1.1 IIOP/SSL considerations

Bi-directional IIOP can operate over IIOP/SSL (*see CORBAservices Chapter 15*) without defining any additions to the IIOP/SSL or the bi-directional GIOP mechanisms. However, if the client wants to authenticate the server when the client receives a callback this cannot be done unless the client has already authenticated the server. This has to be performed during the initial SSL handshake. It is not possible to do this at any point after the initial handshake without establishing a new SSL connection (which defeats the purpose of the bi-directional connections).

15.9 Bi-directional GIOP policy

In GIOP protocol versions 1.0 and 1.1, there are strict rules on which side of a connection can issue what type of messages (for example version 1.0 and 1.1 clients can not issue GIOP reply messages). However, as documented above, it is sensible to relax this restriction if the ORB supports this functionality and policies dictate that bi-directional connection are allowed. To indicate a bi-directional policy, the following is defined.

```
// Self contained module for Bi-directional GIOP policy

module BiDirPolicy {

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = 37;

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```

A **BidirectionalPolicyValue** of **NORMAL** states that the usual GIOP restrictions of who can send what GIOP messages apply (i.e., bi-directional connections are not allowed). A value of **BOTH** indicates that there is a relaxation in what party can issue what GIOP messages (i.e., bi-directional connections are supported). The default value of a **BidirectionalPolicy** is **NORMAL**.

In the absence of a **BidirectionalPolicy** being passed in the **PortableServer::POA::create_POA** operation, a **POA** will assume a policy value of **NORMAL**.

A client and a server **ORB** must each have a **BidirectionalPolicy** with a value of **BOTH** for bi-directional communication to take place.

To create a **BidirectionalPolicy**, the **ORB::create_policy** operation is used.

15.10 OMG IDL

This section contains the OMG IDL for the GIOP and IIOP modules.

15.10.1 GIOP Module

```
module GIOP {       // IDL extended for version 1.1, 1.2, and 1.3

    struct Version {
        octet       major;
        octet       minor;
```

```
    };

#ifndef GIOP_1_1
    // GIOP 1.0
    enum MsgType_1_0{   // rename from MsgType
        Request, Reply,     CancelRequest,
        LocateRequest,      LocateReply,
        CloseConnection,    MessageError
        };

#else
    // GIOP 1.1
    enum MsgType_1_1{
        Request,            Reply,          CancelRequest,
        LocateRequest,      LocateReply,
        CloseConnection,    MessageError,
        Fragment            // GIOP 1.1 addition
    };
#endif

    // GIOP 1.0
    struct MessageHeader_1_0 {// Renamed from MessageHeader
        char                        magic [4];
        Version                     GIOP_version;
        boolean                     byte_order;
        octet                       message_type;
        unsigned long               message_size;
    };

    // GIOP 1.1
    struct MessageHeader_1_1 {
        char                        magic [4];
        Version                     GIOP_version;
        octet                       flags;          // GIOP 1.1 change
        octet                       message_type;
        unsigned long               message_size;
    };

    // GIOP 1.2 and 1.3
    typedef MessageHeader_1_1 MessageHeader_1_2;
    typedef MessageHeader_1_1 MessageHeader_1_3;

    // GIOP 1.0
    struct RequestHeader _1_0 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        sequence <octet>            object_key;
        string                      operation;
        CORBA::OctetSeq             requesting_principal;
    };
```

```
    // GIOP 1.1
    struct RequestHeader_1_1 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        octet                       reserved[3]; // Added in GIOP 1.1
        sequence <octet>            object_key;
        string                      operation;
        CORBA::OctetSeq             requesting_principal;
    };

    // GIOP 1.2, and 1.3
    typedef short                   AddressingDisposition;
    const short                     KeyAddr = 0;
    const short                     ProfileAddr = 1;
    const short                     ReferenceAddr = 2;

    struct IORAddressingInfo {
        unsigned long               selected_profile_index;
        IOP::IOR                    ior;
    };

    union TargetAddress switch (AddressingDisposition) {
        case KeyAddr:               sequence <octet> object_key;
        case ProfileAddr:           IOP::TaggedProfile profile;
        case ReferenceAddr:         IORAddressingInfo ior;
    };

    struct RequestHeader_1_2 {
        unsigned long               request_id;
        octet                       response_flags;
        octet                       reserved[3];
        TargetAddress               target;
        string                      operation;
        // requesting_principal not in GIOP 1.2 and 1.3
        IOP::ServiceContextList     service_context;    // 1.2 change
    };

#ifndef GIOP_1_2
    // GIOP 1.0 and 1.1
    enum ReplyStatusType_1_0 {// Renamed from ReplyStatusType
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    // GIOP 1.0
    struct ReplyHeader_1_0 {// Renamed from ReplyHeader
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
```

```
        ReplyStatusType_1_0       reply_status;
    };

    // GIOP 1.1
    typedef ReplyHeader_1_0 ReplyHeader_1_1;
    // Same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2, and 1.3
    enum ReplyStatusType_1_2 {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD,
        LOCATION_FORWARD_PERM,      // new value for 1.2
        NEEDS_ADDRESSING_MODE       // new value for 1.2
    };

    struct ReplyHeader_1_2 {
        unsigned long               request_id;
        ReplyStatusType_1_2         reply_status;
        IOP::ServiceContextList     service_context;    // 1.2 change
    };
    typedef ReplyHeader_1_2 ReplyHeader_1_3;

#endif // GIOP_1_2
    struct SystemExceptionReplyBody {
        string                  exception_id;
        unsigned long           minor_code_value;
        unsigned long           completion_status;
    };

    struct CancelRequestHeader {
        unsigned long           request_id;
    };

    // GIOP 1.0
    struct LocateRequestHeader_1_0 {
                                // Renamed LocationRequestHeader
        unsigned long           request_id;
        sequence <octet>        object_key;
    };

    // GIOP 1.1
        typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;
        // Same Header contents for 1.0 and 1.1

    // GIOP 1.2 and 1.3
        struct LocateRequestHeader_1_2 {
            unsigned long           request_id;
            TargetAddress           target;
```

```
    };
    typedef LocateRequestHeader_1_2 LocateRequestHeader_1_3;

#ifndef GIOP_1_2
    // GIOP 1.0 and 1.1
    enum LocateStatusType_1_0 {// Renamed from LocateStatusType
            UNKNOWN_OBJECT,
            OBJECT_HERE,
            OBJECT_FORWARD
    };

    // GIOP 1.0
        struct LocateReplyHeader_1_0 {
                                    // Renamed from LocateReplyHeader
            unsigned long               request_id;
            LocateStatusType_1_0        locate_status;
    };

    // GIOP 1.1
    typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;
    // same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2, and 1.3
    enum LocateStatusType_1_2 {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD,
        OBJECT_FORWARD_PERM,            // new value for GIOP 1.2
        LOC_SYSTEM_EXCEPTION,           // new value for GIOP 1.2
        LOC_NEEDS_ADDRESSING_MODE       // new value for GIOP 1.2
    };

    struct LocateReplyHeader_1_2 {
        unsigned long               request_id;
        LocateStatusType_1_2        locate_status;
    };
    typedef LocateReplyHeader_1_2 LocateReplyHeader_1_3;
#endif // GIOP_1_2

    // GIOP 1.2, and 1.3
    struct FragmentHeader_1_2 {
        unsigned long               request_id;
    };
    typedef FragmentHeader_1_2 FragmentHeader_1_3;
};
```

15.10.2 IIOP Module

```
module IIOP {   // IDL extended for version 1.1, 1.2, and 1.3
```

```
    struct Version {
        octet                   major;
        octet                   minor;
    };

    struct ProfileBody_1_0 {// renamed from ProfileBody
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        sequence <octet>        object_key;
    };

    struct ProfileBody_1_1 {// also used for 1.2, and 1.3
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        sequence <octet>        object_key;

        // Added in 1.1 unchanged for 1.2, and 1.3
        sequence <IOP::TaggedComponent> components;
    };

    struct ListenPoint {
        string   host;
        unsigned short port;
    };

    typedef sequence<ListenPoint> ListenPointList;

    struct BiDirIIOPServiceContext {// BI_DIR_IIOP Service Context
        ListenPointList listen_points;
    };
};
```

15.10.3 BiDirPolicy Module

```
// Self contained module for Bi-directional GIOP policy

module BiDirPolicy {

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = 37;

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```