INTERNATIONAL STANDARD

ISO/IEC 19500-2

First edition
2003-04-01

Information technology — Open Distributed Processing —

Part 2: General Inter-ORB Protocol (GIOP)/Internet Inter-ORB Protocol (IIOP)

Technologies de l'information — Traitement réparti ouvert —

Partie 2: «General Inter-ORB Protocol (GIOP)/Internet Inter-ORB Protocol (IIOP)»

Reference number
ISO/IEC 19500-2:2003(E)

© ISO/IEC 2003

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2003

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

© ISO/IEC 2003 — All rights reserved

1 Scope . . . 1
2 Normative references . . . 1
2.1 Identical Recommendations | International Standards . . . 1
2.2 Other Specifications . . . 2
3 Definitions . . . 2
3.1 Recommendations | International Standards . . . 2
3.2 Other Specifications . . . 3
3.2.1 adapter . . . 3
3.2.2 Attribute . . . 3
3.2.3 client . . . 3
3.2.4 data type . . . 3
3.2.5 domain . . . 3
3.2.6 dynamic invocation . . . 3
3.2.7 dynamic skeleton . . . 3
3.2.8 implementation . . . 3
3.2.9 interface repository . . . 3
3.2.10 ORB core . . . 3
3.2.11 repository . . . 4
3.2.12 request . . . 4
3.2.13 results . . . 4
3.2.14 server . . . 4
3.2.15 signature . . . 4
3.2.16 skeleton . . . 4
3.2.17 synchronous request . . . 4
3.2.18 interface type . . . 4
3.2.19 interoperability . . . 4
3.2.20 language binding or mapping . . . 4
3.2.21 method . . . 4
3.2.22 object adapter . . . 5
3.2.23 object implementation . . . 5
3.2.24 object reference . . . 5
3.2.25 objref . . . 5
3.2.26 value . . . 5
3.3 Abbreviations . . . 5
4 Introduction to GIOP/IIOP . . . 6
5 ORB Interoperability Architecture . . . 6
5.1 Overview . . . 6
5.1.1 Domains . . . 6
5.1.2 Bridging Domains . . . 7

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

5.2 ORBs and ORB Services 7
5.2.1 The Nature of ORB Services 7
5.2.2 ORB Services and Object Requests 8
5.2.3 Selection of ORB Services 8
5.3 Domains 9
5.3.1 Definition of a Domain 9
5.3.2 Mapping Between Domains: Bridging 10
5.4 Interoperability Between ORBs 10
5.4.1 ORB Services and Domains 10
5.4.2 ORBs and Domains 11
5.4.3 Interoperability Approaches 11
5.4.4 Policy-Mediated Bridging 13
5.4.5 Configurations of Bridges in Networks 13
5.5 Object Addressing 14
5.5.1 Domain-relative Object Referencing 14
5.5.2 Handling of Referencing Between Domains 15
5.6 An Information Model for Object References 16
5.6.1 What Information Do Bridges Need? 16
5.6.2 Interoperable Object References: IORs 16
5.6.3 Standard IOR Components 19
5.6.4 Profile and Component Composition in IORs 21
5.6.5 IOR Creation and Scope 22
5.6.6 Stringified Object References 22
5.6.7 Object Service Context 23
5.7 Code Set Conversion 27
5.7.1 Character Processing Terminology 27
5.7.2 Code Set Conversion Framework 29
5.7.3 Mapping to Generic Character Environments 35
5.8 Example of Generic Environment Mapping 37
5.8.1 Generic Mappings 37
5.8.2 Interoperation and Generic Mappings 37
5.9 Relevant OSFM Registry Interfaces 37
5.9.1 Character and Code Set Registry 37
5.9.2 Access Routines 38
6 General Inter-ORB Protocol 42
6.1 Goals of the General Inter-ORB Protocol 42
6.1.1 GIOP Overview 43
6.1.2 Common Data Representation (CDR) 43

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.1.3 GIOP Message Overview 44
6.1.4 GIOP Message Transfer 44
6.2 CDR Transfer Syntax 44
6.2.1 Primitive Types 45
6.2.2 OMG IDL Constructed Types 49
6.2.3 Value Types 52
6.2.4 Pseudo-Object Types 59
6.2.5 Object References 64
6.2.6 Abstract Interfaces 64
6.3 GIOP Message Formats 64
6.3.1 GIOP Message Header 65
6.3.2 Request Message 67
6.3.3 Reply Message 70
6.3.4 CancelRequest Message 73
6.3.5 LocateRequest Message 74
6.3.6 LocateReply Message 75
6.3.7 CloseConnection Message 77
6.3.8 MessageError Message 78
6.3.9 Fragment Message 78
6.4 GIOP Message Transport 79
6.4.1 Connection Management 79
6.4.2 Message Ordering 80
6.5 Object Location 81
6.6 Internet Inter-ORB Protocol (IIOP) 82
6.6.1 TCP/IP Connection Usage 82
6.6.2 IIOP IOR Profiles 83
6.6.3 IIOP IOR Profile Components 85
6.7 Bi-Directional GIOP 86
6.7.1 Bi-Directional IIOP 87
6.8 Bi-directional GIOP policy 88
6.9 OMG IDL 89
6.9.1 GIOP Module 89
6.9.2 IIOP Module 95
6.9.3 BiDirPolicy Module 96

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19500-2 was prepared by the Object Management Group (OMG) and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

ISO/IEC 19500 consists of the following parts, under the general title *Information technology — Open Distributed Processing*:

— *Part 2: General Inter-ORB Protocol (GIOP)/Internet Inter-ORB Protocol (IIOP)*

NOTE Other parts will be added in the future.

© ISO/IEC 2003 — All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Introduction

The rapid growth of distributed processing has lead to a need for a coordinating framework for the standardization of Open Distributed Processing (ODP). ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability and portability can be integrated.

Within the framework provided by the RM-ODP, ITU-T Rec. X.931 | ISO/IEC 14752, ODP - Protocol Support for Computational Interactions, defines how interactions between computational objects in a computational specification of a system relate to protocol support for those interactions in an engineering specification of that system.

Annex A to ITU-T Rec. X.931 | ISO/IEC 14752 defines a mapping to the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP) which are specified by this International Standard.

GIOP is the base for all interoperability and support for all object request broker (ORB) functionality in the Common Object Request Broker Architecture (CORBA) specified by the Object Management Group (OMG). IIOP is the mapping of GIOP for the Internet.

Note: This document is technically aligned with the OMG CORBA GIOP and IIOP specifications.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Copyright International Organization for Standardization

Information technology — Open Distributed Processing —

Part 2: General Inter-ORB Protocol (GIOP)/Internet Inter-ORB Protocol (IIOP)

1 Scope

This standard specifies the General Inter-ORB Protocol (GIOP) for object request broker (ORB) interoperability. GIOP can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions defined by this standard.

This standard also defines the Internet Inter-ORB Protocol (IIOP), a specific mapping of the GIOP which runs directly over connections that use the Internet Protocol and the Transmission Control Protocol (TCP/IP connections).

This standard provides a widely implemented and used particularization of ITU-T Rec. X.931 | ISO/IEC 14752, *Information technology — Open Distributed Processing— Protocol support for computational interactions*. It supports interoperability and location transparency in ODP systems.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, *Information technology — Open Distributed Processing — Reference Model: Foundations*
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, *Information technology — Open Distributed Processing — Reference Model: Architecture*
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1999, *Information technology — Open Distributed Processing — Interface Definition Language*
- ISO/IEC 14752:2000, *Information technology — Open Distributed Processing— Protocol support for computational interactions*
- ISO/IEC 8859-1: 1998, *Information technology — 8-bit single-byte coded graphic character sets— Part 1: Latin alphabet No. 1*
- ISO/IEC 10646-1:2000, *Information technology — Universal Multiple-Octect Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

2.2 Other Specifications

- CORBA 2.3 - The Common Object Request Broker: Architecture and Specification, Revision 2.3, Object Management Group, June 1999 (OMG Doc Number: Formal/98-12-01, ftp://ftp.omg.org/pub/docs/formal/98-12-01.pdf)
- CORBAservices: Common Object Services Specification, Object Management Group, December 1998 (OMG Doc Number: Formal/98-12-09, ftp://ftp.omg.org/pub/docs/formal/98-12-09.pdf)
- Java™ to IDL Language Mapping, Object Management Group, July 1999 (OMG Doc Number: Formal/99-07-59, ftp://ftp.omg.org/pub/docs/formal/99-07-59.pdf)
- STD 007 (also, RFC 793), Transmission Control Protocol, J. Postel, Internet Engineering Task Force, Sept. 1981
- STD 005 (also, RFC 791), Internet Protocol, J. Postel, Internet Engineering Task Force, Sept. 1981
- OSF Character and Code Set Registry, OSF DCE FRC 40.1 (Public Version), S. (Martin) O'Donnell, June 1994.
- RPC Runtime Support For I18N Characters — Functional Specification, OSF DCE SIG RFC 41.2, M. Romagna, R. Mackey, November 1994.

3 Definitions

For the purposes of this International Standard, the following definitions apply:

3.1 Recommendations | International Standards

This International Standard makes use of the following terms defined in ITU-T Rec. X.902 | ISO/IEC 10746-2:

behavior

interface

instance

object

state

type

This International Standard makes use of the following terms defined in ITU-T Rec. X.903 | ISO/IEC 10746-3:

operation

stub

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

3.2 Other Specifications

3.2.1 adapter

Same as object adapter.

3.2.2 Attribute

An identifiable association between an object and a value. An attribute **A** is made visable to clients as a pair of operations: **get_A** and **set_A**. Readonly attributes only generate a **get** operation.

3.2.3 client

The code or process that invokes an operation on an object.

3.2.4 data type

A categorization of values operation arguments, typically covering both behavior and representation (i.e., the traditional no-OO programming language notion of type.)

3.2.5 domain

A concept important to interoperability, it is a distinct scope, within which common characteristics are exhibited, common rules observed, and over which a distribution transparency is preserved.

3.2.6 dynamic invocation

Constructing and issuing a request whose signature is possibly not known until run-time.

3.2.7 dynamic skeleton

An interface-independent kind of skeleton, used by servers to handle requests whose signatures are possibly not known until run-time.

3.2.8 implementation

A definition that provides the information needed to create an object and allow the object to participate in providing an appropriate set of services. An implementation typically includes a description of the data structure used to represent the core state associated with an object, as well as definitions of the methods that access that data structure. It will also typically include information about the intended interface of the object.

3.2.9 interface repository

A storage place for interface information.

3.2.10 ORB core

The ORB component which moves a request from a client to the appropriate adapter for the target object.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

3.2.11 repository

See interface repository and implementation repository.

3.2.12 request

A client issues a request to cause a service to be perfomed. A request consists of an operation and zero or more actual parameters.

3.2.13 results

The information returned to the client, which may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

3.2.14 server

A process implementing one or more operations on one or more objects.

3.2.15 signature

Defines the parameters of a given operation including their number order, data types, and passing mode; the results if any; and the possible outcomes (normal vs. exceptional) that might occur.

3.2.16 skeleton

The object-interface-specific ORB component which assists an object adapter in passing requests to particular methods.

3.2.17 synchronous request

A request where the client pauses to wiat for completion of the request. Contrast with deferred synchronous request and one-way request.

3.2.18 interface type

A type satisfied by any object that satisfies a particular interface.

3.2.19 interoperability

The ability for tow or more ORBs to cooperate to deliver requests to the proper object. Interoperating ORBs appear to a client to be a single ORB.

3.2.20 language binding or mapping

The means and conventions by which a programmer writing in a specific programming language accesses ORB capabilities.

3.2.21 method

An implementation of an operation. Code that may be executed to perform a requested service. Methods associated with an object may be structured into one or more programs.

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

3.2.22 object adapter

The ORB component which provides object reference, activation, and state related services to an object implementation. There may be different adapters provided for different kinds of implementations.

3.2.23 object implementation

Same as implementation.

3.2.24 object reference

A value that unambiguously identifies an object. Object references are never reused to identify another object.

3.2.25 objref

An abbreviation for object reference.

3.2.26 value

Any entity that may be a possible actual parameter in a request. Values that serve to identify objects are called object references.

3.3 Abbreviations

For the purposes of this International Standard, the following abbreviations apply:

ADT	Abstract Data Type
CCCS	Client Conversion Code Sets
CCS	Conversion Code Sets
CDR	Common Data Representation
CMIR	Client Makes it Right
CNCS	Client Native Code Set
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
OMG	Object Management Group
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
ORB	Object Request Broker
SCCS	Server Conversion Code Sets
SMIR	Server Makes It Right

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

SNCS	Server Native Code Set
TCS	Transmission Code Set
TCS-C	Char Transmission Code Set
TCS-W	Wchar Transmission Code Set
VSCID	Vender Service Context codeset ID

4 Introduction to GIOP/IIOP

This standard specifies the General Inter-ORB Protocol (GIOP) for object request broker (ORB) interoperability. GIOP can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions. This standard 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 its implementation. Such support does not require using it internally; conforming ORBs may also provide bridges to this protocol.

A definition of interoperability material necessary for the protocol is in clause 5; the definition of GIOP/IIOP is in clause 6. This standard is technically aligned with sections 13 and 15 of CORBA 2.3.

5 ORB Interoperability Architecture

For the purposes of this standard interoperability is defined as the ability for a client on ORB A to invode an IDL-defined operation on an object on ORB B, where ORB A and ORB B are independently developed. It further identifies general requirements including in particular:

- Ability for two vendors' ORBs to interoperate without prior knowledge of each other's implementations;
- Support of all ORB functionality;
- Preservation of content and semantics of ORB-specific information across ORB boundaries (for example, security).

In effect, the requirement is for invocations between client and server objects to be independent of whether they are on the same or different ORBs, and not to mandate fundamental modifications to existing ORB products.

5.1 Overview

5.1.1 Domains

The CORBA Object Model identifies various distribution transparencies that must be supported within a single ORB environment, such as location transparency. Elements of ORB functionality often correspond directly to such transparencies. Interoperability can be viewed as extending transparencies to span multiple ORBs.

In this architecture a *domain* is a distinct scope, within which certain common characteristics are exhibited and common rules are observed over which a distribution transparency is preserved. Thus, interoperability is fundamentally involved with transparently crossing such domain boundaries.

Domains tend to be either administrative or technological in nature, and need not correspond to the boundaries of an ORB installation. Administrative domains include naming domains, trust groups, resource management domains and other "run-time" characteristics of a system. Technology domains identify common protocols, syntaxes and similar

 © ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

"build-time" characteristics. In many cases, the need for technology domains derives from basic requirements of administrative domains.

Within a single ORB, most domains are likely to have similar scope to that of the ORB itself: common object references, network addresses, security mechanisms, and more. However, it is possible for there to be multiple domains of the same type supported by a given ORB: internal representation on different machine types, or security domains. Conversely, a domain may span several ORBs: similar network addresses may be used by different ORBs, type identifiers may be shared.

5.1.2 Bridging Domains

The abstract architecture describes ORB interoperability in terms of the translation required when an object request traverses domain boundaries. Conceptually, a mapping or *bridging mechanism* resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain.

The concrete architecture identifies two approaches to inter-ORB bridging:

- At application level, allowing flexibility and portability.
- At ORB level, built into the ORB itself.

5.2 ORBs and ORB Services

The ORB Core is that part of the ORB which provides the basic representation of objects and the communication of requests. The ORB Core therefore supports the minimum functionality to enable a client to invoke an operation on a server object, with (some of) the distribution transparencies required by *CORBA*.

An object request may have implicit attributes which affect the way in which it is communicated - though not the way in which a client makes the request. These attributes include security, transactional capabilities, recovery, and replication. These features are provided by "ORB Services," which will in some ORBs be layered as internal services over the core, or in other cases be incorporated directly into an ORB's core. It is an aim of this specification to allow for new ORB Services to be defined in the future, without the need to modify or enhance this architecture.

Within a single ORB, ORB services required to communicate a request will be implemented and (implicitly) invoked in a private manner. For interoperability between ORBs, the ORB services used in the ORBs, and the correspondence between them, must be identified.

5.2.1 The Nature of ORB Services

ORB Services are invoked implicitly in the course of application-level interactions. ORB Services range from fundamental mechanisms such as reference resolution and message encoding to advanced features such as support for security, transactions, or replication.

An ORB Service is often related to a particular transparency. For example, message encoding – the marshaling and unmarshaling of the components of a request into and out of message buffers – provides transparency of the representation of the request. Similarly, reference resolution supports location transparency. Some transparencies, such as security, are supported by a combination of ORB Services and Object Services while others, such as replication, may involve interactions between ORB Services themselves.

ORB Services differ from Object Services in that they are positioned below the application and are invoked transparently to the application code. However, many ORB Services include components which correspond to conventional Object Services in that they are invoked explicitly by the application.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Security is an example of service with both ORB Service and normal Object Service components, the ORB components being those associated with transparently authenticating messages and controlling access to objects while the necessary administration and management functions resemble conventional Object Services.

5.2.2 ORB Services and Object Requests

Interoperability between ORBs extends the scope of distribution transparencies and other request attributes to span multiple ORBs. This requires the establishment of relationships between supporting ORB Services in the different ORBs.

In order to discuss how the relationships between ORB Services are established, it is necessary to describe an abstract view of how an operation invocation is communicated from client to server object.

1. The client generates an operation request, using a reference to the server object, explicit parameters, and an implicit invocation context. This is processed by certain ORB Services on the client path.
2. On the server side, corresponding ORB Services process the incoming request, transforming it into a form directly suitable for invoking the operation on the server object.
3. The server object performs the requested operation.
4. Any result of the operation is returned to the client in a similar manner.

The correspondence between client-side and server-side ORB Services need not be one-to-one and in some circumstances may be far more complex. For example, if a client application requests an operation on a replicated server, there may be multiple server-side ORB service instances, possibly interacting with each other.

In other cases, such as security, client-side or server-side ORB Services may interact with Object Services such as authentication servers.

5.2.3 Selection of ORB Services

The ORB Services used are determined by:

- Static properties of both client and server objects; for example, whether a server is replicated.
- Dynamic attributes determined by a particular invocation context; for example, whether a request is transactional.
- Administrative policies (e.g., security).

Within a single ORB, private mechanisms (and optimizations) can be used to establish which ORB Services are required and how they are provided. Service selection might in general require negotiation to select protocols or protocol options. The same is true between different ORBs: it is necessary to agree which ORB Services are used, and how each transforms the request. Ultimately, these choices become manifest as one or more protocols between the ORBs or as transformations of requests.

In principle, agreement on the use of each ORB Service can be independent of the others and, in appropriately constructed ORBs, services could be layered in any order or in any grouping. This potentially allows applications to specify selective transparencies according to their requirements, although at this time CORBA provides no way to penetrate its transparencies.

A client ORB must be able to determine which ORB Services must be used in order to invoke operations on a server object. Correspondingly, where a client requires dynamic attributes to be associated with specific invocations, or administrative policies dictate, it must be possible to cause the appropriate ORB Services to be used on client and

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

server sides of the invocation path. Where this is not possible - because, for example, one ORB does not support the full set of services required - either the interaction cannot proceed or it can only do so with reduced facilities or transparencies.

5.3 Domains

From a computational viewpoint, the OMG Object Model identifies various distribution transparencies which ensure that client and server objects are presented with a uniform view of a heterogeneous distributed system. From an engineering viewpoint, however, the system is not wholly uniform. There may be distinctions of location and possibly many others such as processor architecture, networking mechanisms and data representations. Even when a single ORB implementation is used throughout the system, local instances may represent distinct, possibly optimized scopes for some aspects of ORB functionality.

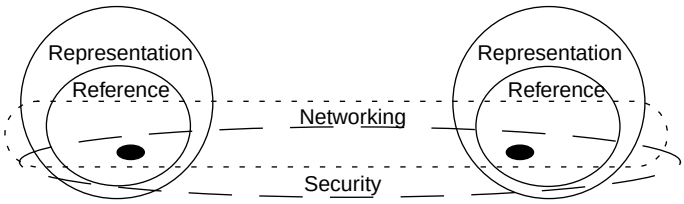


Figure 1 - Different Kinds of Domains can Coexist

Interoperability, by definition, introduces further distinctions, notably between the scopes associated with each ORB. To describe both the requirements for interoperability and some of the solutions, this architecture introduces the concept of *domains* to describe the scopes and their implications.

Informally, a domain is a set of objects sharing a common characteristic or abiding by common rules. It is a powerful modelling concept which can simplify the analysis and description of complex systems. There may be many types of domains (e.g., management domains, naming domains, language domains, and technology domains).

5.3.1 Definition of a Domain

Domains allow partitioning of systems into collections of components which have some characteristic in common. In this architecture a domain is a scope in which a collection of objects, said to be members of the domain, is associated with some common characteristic; any object for which the association does not exist, or is undefined, is not a member of the domain. A domain can be modelled as an object and may be itself a member of other domains.

It is the scopes themselves and the object associations or bindings defined within them which characterize a domain. This information is disjoint between domains. However, an object may be a member of several domains, of similar kinds as well as of different kinds, and so the sets of members of domains may overlap.

The concept of a domain boundary is defined as the limit of the scope in which a particular characteristic is valid or meaningful. When a characteristic in one domain is translated to an equivalent in another domain, it is convenient to consider it as traversing the boundary between the two domains.

Domains are generally either administrative or technological in nature. Examples of domains related to ORB interoperability issues are:

- Referencing domain – the scope of an object reference
- Representation domain – the scope of a message transfer syntax and protocol

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

- Network addressing domain – the scope of a network address
- Network connectivity domain – the potential scope of a network message
- Security domain – the extent of a particular security policy
- Type domain – the scope of a particular type identifier
- Transaction domain – the scope of a given transaction service

Domains can be related in two ways: containment, where a domain is contained within another domain, and federation, where two domains are joined in a manner agreed to and set up by their administrators.

5.3.2 Mapping Between Domains: Bridging

Interoperability between domains is only possible if there is a well-defined mapping between the behaviors of the domains being joined. Conceptually, a mapping mechanism or bridge resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain. Note that the use of the term "bridge" in this context is conceptual and refers only to the functionality which performs the

required mappings between distinct domains. There are several implementation options for such bridges and these are discussed elsewhere.

For full interoperability, it is essential that all the concepts used in one domain are transformable into concepts in other domains with which interoperability is required, or that if the bridge mechanism filters such a concept out, nothing is lost as far as the supported objects are concerned. In other words, one domain may support a superior service to others, but such a superior functionality will not be available to an application system spanning those domains.

A special case of this requirement is that the object models of the two domains need to be compatible. This specification assumes that both domains are strictly compliant with the CORBA Object Model and the *CORBA* specifications. This includes the use of OMG IDL (technically aligned with ISO/IEC 14750) when defining interfaces, the use of the CORBA Core Interface Repository, and other modifications that were made to *CORBA*. Variances from this model could easily compromise some aspects of interoperability.

5.4 Interoperability Between ORBs

An ORB "provides the mechanisms by which objects transparently make and receive requests and responses. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments..." ORB interoperability extends this definition to cases in which client and server objects on different ORBs "transparently make and receive requests..."

Note that a direct consequence of this transparency requirement is that bridging must be bidirectional: that is, it must work as effectively for object references passed as parameters as for the target of an object invocation. Were bridging unidirectional (e.g., if one ORB could only be a client to another) then transparency would not have been provided, because object references passed as parameters would not work correctly: ones passed as "callback objects," for example, could not be used.

Without loss of generality, most of this specification focuses on bridging in only one direction. This is purely to simplify discussions, and does not imply that unidirectional connectivity satisfies basic interoperability requirements.

5.4.1 ORB Services and Domains

In this architecture, different aspects of ORB functionality - ORB Services - can be considered independently and associated with different domain types. The architecture does not, however, prescribe any particular decomposition

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

of ORB functionality and interoperability into ORB Services and corresponding domain types. There is a range of possibilities for such a decomposition:

1. The simplest model, for interoperability, is to treat all objects supported by one ORB (or, alternatively, all ORBs of a given type) as comprising one domain. Interoperability between any pair of different domains (or domain types) is then achieved by a specific all-encompassing bridge between the domains. (This is all *CORBA* implies.)

2. More detailed decompositions would identify particular domain types - such as referencing, representation, security, and networking. A core set of domain types would be pre-determined and allowance made for additional domain types to be defined as future requirements dictate (e.g., for new ORB Services).

5.4.2 ORBs and Domains

In many respects, issues of interoperability between ORBs are similar to those which can arise with a single type of ORB (e.g., a product). For example:

- Two installations of the ORB may be installed in different security domains, with different Principal identifiers. Requests crossing those security domain boundaries will need to establish locally meaningful Principals for the caller identity, and for any Principals passed as parameters.

- Different installations might assign different type identifiers for equivalent types, and so requests crossing type domain boundaries would need to establish locally meaningful type identifiers (and perhaps more).

Conversely, not all of these problems need to appear when connecting two ORBs of a different type (e.g., two different products). Examples include:

- They could be administered to share user visible naming domains, so that naming domains do not need bridging.

- They might reuse the same networking infrastructure, so that messages could be sent without needing to bridge different connectivity domains.

Additional problems can arise with ORBs of different types. In particular, they may support different concepts or models, between which there are no direct or natural mappings. CORBA only specifies the application level view of object interactions, and requires that distribution transparencies conceal a whole range of lower level issues. It follows that within any particular ORB, the mechanisms for supporting transparencies are not visible at the application-level and are entirely a matter of implementation choice. So there is no guarantee that any two ORBs support similar internal models or that there is necessarily a straightforward mapping between those models.

These observations suggest that the concept of an ORB (instance) is too coarse or superficial to allow detailed analysis of interoperability issues between ORBs. Indeed, it becomes clear that an ORB instance is an elusive notion: it can perhaps best be characterized as the intersection or coincidence of ORB Service domains.

5.4.3 Interoperability Approaches

When an interaction takes place across a domain boundary, a mapping mechanism, or bridge, is required to transform relevant elements of the interaction as they traverse the boundary. There are essentially two approaches to achieving this: mediated bridging and immediate bridging. These approaches are described in the following subsections.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

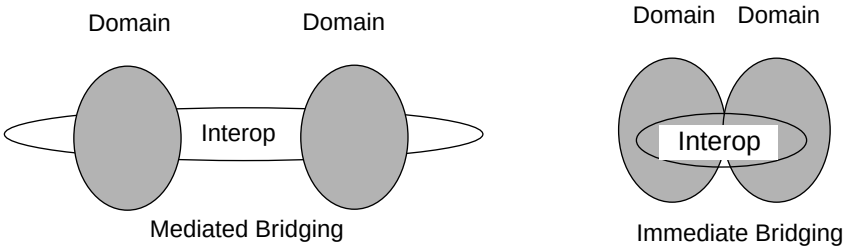


Figure 2 - Two bridging techniques, different uses of an intermediate form agreed on between the two domains.

5.4.3.1 Mediated Bridging

With mediated bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, between the internal form of that domain and an agreed, common form.

Observations on mediated bridging are as follows:

- The scope of agreement of a common form can range from a private agreement between two particular ORB/domain implementations to a universal standard.
- There can be more than one common form, each oriented or optimized for a different purpose.
- If there is more than one possible common form, then which is used can be static (e.g., administrative policy agreed between ORB vendors, or between system administrators) or dynamic (e.g., established separately for each object, or on each invocation).
- Engineering of this approach can range from in-line specifically compiled (compare to stubs) or generic library code (such as encryption routines), to intermediate bridges to the common form.

5.4.3.2 Immediate Bridging

With immediate bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, directly between the internal form of one domain and the internal form of the other.

Observations on immediate bridging are as follows:

- This approach has the potential to be optimal (in that the interaction is not mediated via a third party, and can be specifically engineered for each pair of domains) but sacrifices flexibility and generality of interoperability to achieve this.
- This approach is often applicable when crossing domain boundaries which are purely administrative (i.e., there is no change of technology). For example, when crossing security administration domains between similar ORBs, it is not necessary to use a common intermediate standard.

As a general observation, the two approaches can become almost indistinguishable when private mechanisms are used between ORB/domain implementations.

5.4.3.3 Location of Inter-Domain Functionality

Logically, an inter-domain bridge has components in both domains, whether the mediated or immediate bridging approach is used. However, domains can span ORB boundaries and ORBs can span machine and system boundaries;

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

conversely, a machine may support, or a process may have access to more than one ORB (or domain of a given type). From an engineering viewpoint, this means that the components of an inter-domain bridge may be dispersed or co-located, with respect to ORBs or systems. It also means that the distinction between an ORB and a bridge can be a matter of perspective: there is a duality between viewing inter-system messaging as belonging to ORBs, or to bridges.

For example, if a single ORB encompasses two security domains, the inter-domain bridge could be implemented wholly within the ORB and thus be invisible as far as ORB interoperability is concerned. A similar situation arises when a bridge between two ORBs or domains is implemented wholly within a process or system which has access to both. In such cases, the engineering issues of inter-domain bridging are confined, possibly to a single system or process. If it were practical to implement all bridging in this way, then interactions between systems or processes would be solely within a single domain or ORB.

5.4.3.4 Bridging Level

As noted at the start of this section, bridges may be implemented both internally to an ORB and as layers above it. These are called respectively “in-line” and “request-level” bridges.

Request-level bridges use the CORBA APIs, including the Dynamic Skeleton Interface, to receive and issue requests. However, there is an emerging class of “implicit context” which may be associated with some invocations, holding ORB Service information such as transaction and security context information, which is not at this time exposed through general purpose public APIs. (Those APIs expose only OMG IDL-defined operation parameters, not implicit ones.) Rather, the precedent set with the Transaction Service is that special purpose APIs are defined to allow bridging of each kind of context. This means that request-level bridges must be built to specifically understand the implications of bridging such ORB Service domains, and to make the appropriate API calls.

5.4.4 Policy-Mediated Bridging

An assumption made through most of this specification is that the existence of domain boundaries should be transparent to requests: that the goal of interoperability is to hide such boundaries. However, if this were always the goal, then there would be no real need for those boundaries in the first place.

Realistically, administrative domain boundaries exist because they reflect ongoing differences in organizational policies or goals. Bridging the domains will in such cases require *policy mediation*. That is, inter-domain traffic will need to be constrained, controlled, or monitored; fully transparent bridging may be highly undesirable. Resource management policies may even need to be applied, restricting some kinds of traffic during certain periods.

Security policies are a particularly rich source of examples: a domain may need to audit external access, or to provide domain-based access control. Only a very few objects, types of objects, or classifications of data might be externally accessible through a “firewall.”

Such policy-mediated bridging requires a bridge that knows something about the traffic being bridged. It could in general be an application-specific policy, and many policy-mediated bridges could be parts of applications. Those might be organization-specific, off-the-shelf, or anywhere in between.

Request-level bridges, which use only public ORB APIs, easily support the addition of policy mediation components, without loss of access to any other system infrastructure that may be needed to identify or enforce the appropriate policies.

5.4.5 Configurations of Bridges in Networks

In the case of network-aware ORBs, we anticipate that some ORB protocols will be more frequently bridged to than others, and so will begin to serve the role of “backbone ORBs.” (This is a role that the IIOP is specifically expected to serve.) This use of “backbone topology” is true both on a large scale and a small scale. While a large scale public data

© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

network provider could define its own backbone ORB, on a smaller scale, any given institution will probably designate one commercially available ORB as its backbone.

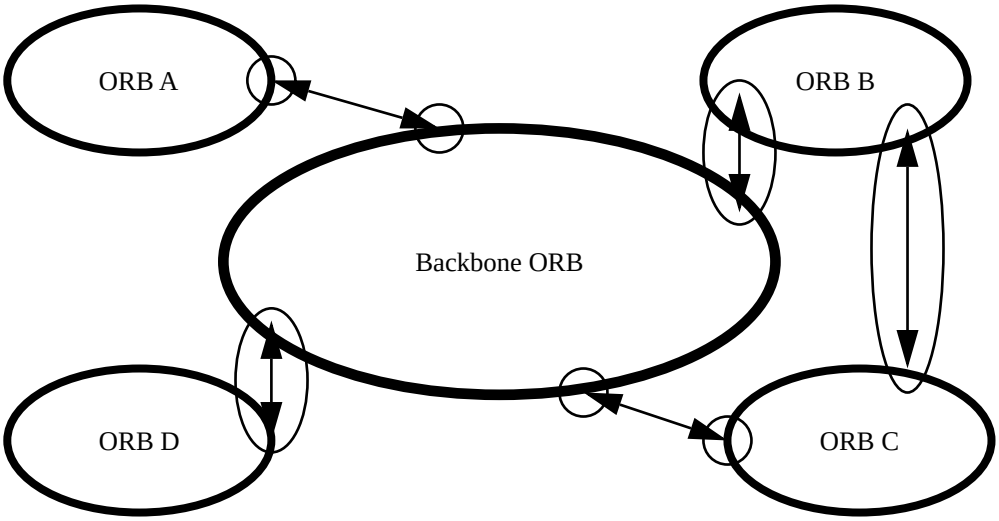


Figure 3 - An ORB chosen as a backbone will connect other ORBs through bridges, both full-bridges and half-bridges.

Adopting a backbone style architecture is a standard administrative technique for managing networks. It has the consequence of minimizing the number of bridges needed, while at the same time making the ORB topology match typical network organizations. (That is, it allows the number of bridges to be proportional to the number of protocols, rather than combinatorial.)

In large configurations, it will be common to notice that adding ORB bridges doesn't even add any new "hops" to network routes, because the bridges naturally fit in locations where connectivity was already indirect, and augment or supplant the existing network firewalls.

5.5 Object Addressing

The CORBA Object Model defines an object reference as an object name that reliably denotes a particular object. An object reference identifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references.

The fundamental ORB interoperability requirement is to allow clients to use such object names to invoke operations on objects in other ORBs. Clients do not need to distinguish between references to objects in a local ORB or in a remote one. Providing this transparency can be quite involved, and naming models are fundamental to it.

This section of this specification discusses models for naming entities in multiple domains, and transformations of such names as they cross the domain boundaries. That is, it presents transformations of object reference information as it passes through networks of inter-ORB bridges. It uses the word "ORB" as synonymous with referencing domain; this is purely to simplify the discussion. In other contexts, "ORB" can usefully denote other kinds of domain.

5.5.1 Domain-relative Object Referencing

Since CORBA does not require ORBs to understand object references from other ORBs, when discussing object references from multiple ORBs one must always associate the object reference's domain (ORB) with the object

© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

reference. We use the notation *D0.R0* to denote an object reference *R0* from domain *D0*; this is itself an object reference. This is called "domain-relative" referencing (or addressing) and need not reflect the implementation of object references within any ORB.

At an implementation level, associating an object reference with an ORB is only important at an inter-ORB boundary; that is, inside a bridge. This is simple, since the bridge knows from which ORB each request (or response) came, including any object references embedded in it.

5.5.2 Handling of Referencing Between Domains

When a bridge hands an object reference to an ORB, it must do so in a form understood by that ORB: the object reference must be in the recipient ORB's native format. Also, in cases where that object originated from some other ORB, the bridge must associate each newly created "proxy" object reference with (what it sees as) the original object reference.

Several basic schemes to solve these two problems exist. These all have advantages in some circumstances; all can be used, and in arbitrary combination with each other, since CORBA object references are opaque to applications. The ramifications of each scheme merits attention, with respect to scaling and administration. The schemes include:

1. *Object Reference Translation Reference Embedding*: The bridge can store the original object reference itself, and pass an entirely different proxy reference into the new domain. The bridge must then manage state on behalf of each bridged object reference, map these references from one ORB's format to the other's, and vice versa.

2. *Reference Encapsulation:* The bridge can avoid holding any state at all by conceptually concatenating a domain identifier to the object name. Thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1... D4* it could be identified in *D4* as proxy reference *d3.d2.d1.d0.R*, where *dn* is the address of *Dn* relative to *Dn+1*.

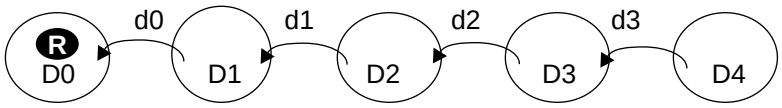


Figure 4 - Reference encapsulation adds domain information during bridging

3. *Domain Reference Translation:* Like object reference translation, this scheme holds some state in the bridge. However, it supports sharing that state between multiple object references by adding a domain-based route identifier to the proxy (which still holds the original reference, as in the reference encapsulation scheme). It achieves this by providing encoded domain route information each time a domain boundary is traversed; thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1...D4* it would be identified in *D4* as *(d3, x3).R*, and in *D2* as *(d1,x1).R*, and so on, where *dn* is the address of *Dn* relative to *Dn+1*, and *xn* identifies the pair *(dn-1, xn-1)*.

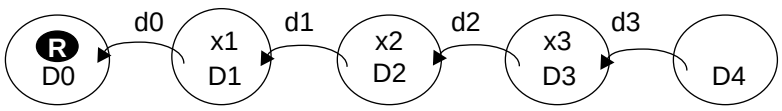


Figure 5 - Domain Reference Translation *substitutes domain references* during bridging

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

4. *Reference Canonicalization:* This scheme is like domain reference translation, except that the proxy uses a "well-known" (e.g., global) domain identifier rather than an encoded path. Thus a reference *R*, originating in domain *D0* would be identified in other domains as *D0.R*.

Observations about these approaches to inter-domain reference handling are as follows:

- Naive application of reference encapsulation could lead to arbitrarily large references. A "topology service" could optimize cycles within any given encapsulated reference and eliminate the appearance of references to local objects as alien references.
- A topology service could also optimize the chains of routes used in the domain reference translation scheme. Since the links in such chains are re-used by any path traversing the same sequence of domains, such optimization has particularly high leverage.
- With the general purpose APIs defined in *CORBA*, object reference translation can be supported even by ORBs not specifically intended to support efficient bridging, but this approach involves the most state in intermediate bridges. As with reference encapsulation, a topology service could optimize individual object references. (APIs are defined by the Dynamic Skeleton Interface and Dynamic Invocation Interface)
- The chain of addressing links established with both object and domain reference translation schemes must be represented as state within the network of bridges. There are issues associated with managing this state.
- Reference canonicalization can also be performed with managed hierarchical name spaces such as those now in use on the Internet and X.500 naming.

5.6 An Information Model for Object References

This section provides a simple, powerful information model for the information found in an object reference. That model is intended to be used directly by developers of bridging technology, and is used in that role by the IIOP, described in the 6.2.5, 'Object References'.

5.6.1 What Information Do Bridges Need?

The following potential information about object references has been identified as critical for use in bridging technologies:

- *Is it null?* Nulls only need to be transmitted and never support operation invocation.
- *What type is it?* Many ORBs require knowledge of an object's type in order to efficiently preserve the integrity of their type systems.
- *What protocols are supported?* Some ORBs support objrefs that in effect live in multiple referencing domains, to allow clients the choice of the most efficient communications facilities available.
- *What ORB Services are available?* As noted in 5.2.3, 'Selection of ORB Services', several different ORB Services might be involved in an invocation. Providing information about those services in a standardized way could in many cases reduce or eliminate negotiation overhead in selecting them.

5.6.2 Interoperable Object References: IORs

To provide the information above, an "Interoperable Object Reference," (IOR) data structure has been provided. This data structure need not be used internally to any given ORB, and is not intended to be visible to application-level ORB programmers. It should be used only when crossing object reference domain boundaries, within bridges.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

This data structure is designed to be efficient in typical single-protocol configurations, while not penalizing multiprotocol ones.

```
module IOP {                                    // IDL

    // Standard Protocol Profile tag values

    typedef unsigned long       ProfileId;
    const ProfileId                 TAG_INTERNET_IOP = 0;
    const ProfileId                 TAG_MULTIPLE_COMPONENTS = 1;

    struct TaggedProfile {
      ProfileId                     tag;
      sequence <octet>              profile_data;
    };

    // an Interoperable Object Reference is a sequence of
    // object-specific protocol profiles, plus a type ID.

    struct IOR {
      string                        type_id;
      sequence <TaggedProfile>      profiles;
    };

    // Standard way of representing multicomponent profiles.
    // This would be encapsulated in a TaggedProfile.

    typedef unsigned long ComponentId;
    struct TaggedComponent {
      ComponentId                   tag;
      sequence <octet>              component_data;
    };
    typedef sequence <TaggedComponent>   MultipleComponentProfile;
};
```

Object references have at least one *tagged profile*. Each profile supports one or more protocols and encapsulates all the basic information the protocols it supports need to identify an object. Any single profile holds enough information

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

to drive a complete invocation using any of the protocols it supports; the content and structure of those profile entries are wholly specified by these protocols. A bridge between two domains may need to know the detailed content of the profile for those domains' profiles, depending on the technique it uses to bridge the domains[1].

Each profile has a unique numeric tag, assigned by the OMG. The ones defined here are for the IIOP (see 6.6.3, 'IIOP IOR Profile Components') and for use in "multiple component profiles." Profile tags in the range **0x80000000** through **0xffffffff** are reserved for future use, and are not currently available for assignment.

Null object references are indicated by an empty set of profiles, and by a "Null" type ID (a string which contains only a single terminating character). A Null TypeID is the only mechanism that can be used to represent the type **CORBA::Object**. Type IDs may only be "Null" in any message, requiring the client to use existing knowledge or to consult the object, to determine interface types supported. The type ID is a Repository ID identifying the interface type, and is provided to allow ORBs to preserve strong typing. This identifier is agreed on within the bridge and, for reasons outside the scope of this interoperability specification, needs to have a much broader scope to address various problems in system evolution and maintenance. Type IDs support detection of type equivalence, and in conjunction with an Interface Repository, allow processes to reason about the relationship of the type of the object referred to and any other type.

The type ID, if provided by the server, indicates the most derived type that the server wishes to publish, at the time the reference is generated. The object's actual most derived type may later change to a more derived type. Therefore, the type ID in the IOR can only be interpreted by the client as a hint that the object supports at least the indicated interface. The client can succeed in narrowing the reference to the indicated interface, or to one of its base interfaces,

based solely on the type ID in the IOR, but must not fail to narrow the reference without consulting the object via the "_is_a" or "_get_interface" pseudo-operations.

5.6.2.1 The TAG_INTERNET_IOP Profile

The **TAG_INTERNET_IOP** tag identifies profiles that support the Internet Inter-ORB Protocol. The **ProfileBody** of this profile, described in detail in 6.6.2, 'IIOP IOR Profiles', contains a CDR encapsulation of a structure containing addressing and object identification information used by IIOP. Version 1.1 of the **TAG_INTERNET_IOP** profile also includes a **sequence<TaggedComponent>** that can contain additional information supporting optional IIOP features, ORB services such as security, and future protocol extensions.

Protocols other than IIOP (such as ESIOPs and other GIOPs) can share profile information (such as object identity or security information) with IIOP by encoding their additional profile information as components in the **TAG_INTERNET_IOP** profile. All **TAG_INTERNET_IOP** profiles support IIOP, regardless of whether they also support additional protocols. Interoperable ORBs are not required to create or understand any other profile, nor are they required to create or understand any of the components defined for other protocols that might share the **TAG_INTERNET_IOP** profile with IIOP.

5.6.2.2 The TAG_MULTIPLE_COMPONENTS Profile

The **TAG_MULTIPLE_COMPONENTS** tag indicates that the value encapsulated is of type **MultipleComponentProfile**. In this case, the profile consists of a list of protocol components, indicating ORB services accessible using that protocol. ORB services are assigned component identifiers in a namespace that is distinct from the profile identifiers. Note that protocols may use the **MultipleComponentProfile** data structure to hold

---

1. Based on topology and policy information available to it, a bridge may find it prudent to add or remove some profiles as it forwards an object reference. For example, a bridge acting as a firewall might remove all profiles except ones that make such profiles, letting clients that understand the profiles make routing choices.

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

profile components even without using **TAG_MULTIPLE_COMPONENTS** to indicate that particular protocol profile, and need not use a **MultipleComponentProfile** to hold sets of profile components.

5.6.2.3 IOR Components

TaggedComponents contained in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles are identified by unique numeric tags using a namespace distinct form that is used for profile tags. Component tags are assigned by the OMG.

Specifications of components must include the following information:

- *Component ID:* The compound tag that is obtained from OMG.
- *Structure and encoding:* The syntax of the component data and the encoding rules. If the component value is encoded as a CDR encapsulation, the IDL type that is encapsulated and the GIOP version which is used for encoding the value, if different than GIOP 1.0, must be specified as part of the component definition.
- *Semantics:* How the component data is intended to be used.
- *Protocols:* The protocol for which the component is defined, and whether it is intended that the component be usable by other protocols.
- *At most once:* whether more than one instance of this component can be included in a profile.

Specification of protocols must describe how the components affect the protocol. The following should be specified in any protocol definition for each **TaggedComponent** that the protocol uses:

- *Mandatory presence:* Whether inclusion of the component in profiles supporting the protocol is required (MANDATORY PRESENCE) or not required (OPTIONAL PRESENCE).
- *Droppable:* For optional presence component, whether component, if present, must be retained or may be dropped.

5.6.3 Standard IOR Components

The following are standard IOR components that can be included in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles, and may apply to IIOP, other GIOPs, ESIOPs, or other protocols. An ORB must not drop these components from an existing IOR.

```
module IOP {
    const ComponentId       TAG_ORB_TYPE = 0;
    const ComponentId       TAG_CODE_SETS = 1;
    const ComponentId       TAG_POLICIES = 2;
    const ComponentId       TAG_ALTERNATE_IIOP_ADDRESS = 3;
    const ComponentId       TAG_ASSOCIATION_OPTIONS = 13;
    const ComponentId       TAG_SEC_NAME = 14;
    const ComponentId       TAG_SPKM_1_SEC_MECH = 15;
    const ComponentId       TAG_SPKM_2_SEC_MECH = 16;
    const ComponentId       TAG_KerberosV5_SEC_MECH = 17;
```

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

```
    const ComponentId       TAG_CSI_ECMA_Secret_SEC_MECH = 18;
    const ComponentId       TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;
    const ComponentId       TAG_SSL_SEC_TRANS = 20;
    const ComponentId       TAG_CSI_ECMA_Public_SEC_MECH = 21;
    const ComponentId       TAG_ GENERIC_SEC_MECH = 22;
    const ComponentId       TAG_JAVA_CODEBASE = 25;
};
```

The following additional components that can be used by other protocols are specified in the DCE ESIOP chapter of *CORBA 2.3* and others in the section on Security Service for DCE ESIOP in the Security Service chapter of *CORBAServices*:

```
    const ComponentId       TAG_COMPLETE_OBJECT_KEY = 5;
    const ComponentId       TAG_ENDPOINT_ID_POSITION = 6;
    const ComponentId       TAG_LOCATION_POLICY = 12;
    const ComponentId       TAG_DCE_STRING_BINDING = 100;
    const ComponentId       TAG_DCE_BINDING_NAME = 101;
    const ComponentId       TAG_DCE_NO_PIPES = 102;
    const ComponentId       TAG_DCE_SEC_MECH = 103; // Security Service
```

5.6.3.1 TAG_ORB_TYPE Component

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

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

The **TAG_ORB_TYPE** component can appear at most once in any IOR profile. For profiles supporting IIOP 1.1 or greater, it is optionally present and may not be dropped.

5.6.3.2 TAG_ALTERNATE_IIOP_ADDRESS Component

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

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

```
struct {
  string HostID,
  short Port
```


© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

};

encoded as a CDR encapsulation.

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

5.6.3.3 Other Components

The following standard components are specified in various OMG specifications. Refer to 2.2, 'Other Specifications' for additional information.

- **TAG_CODE_SETS** (see 5.7.2.4, 'CodeSet Component of IOR Multi-Component Profile').
- **TAG_POLICIES** (reserved for CORBA Messaging specification,which will be incorporated into CORBA 3.0).
- **TAG_SEC_NAME** (see 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_ASSOCIATION_OPTIONS** (see 15.10.3 Tag Association Options, Security chapter - CORBAServices).
- **TAG_SSL_SEC_TRANS** (see 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_GENERIC_SEC_MECH** and all other tags with names in the form **TAG_*_SEC_MECH** (see 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_JAVA_CODEBASE** (see 1.4.9.3, "Codebase Transmission" - Java to IDL Language Mapping).
- **TAG_COMPLETE_OBJECT_KEY (see 16.5.4, "Complete Object Key Component" - CORBA 2.3).**
- **TAG_ENDPOINT_ID_POSITION (see 16.5.5, "Endpoint ID Position Component" - CORBA 2.3).**
- **TAG_LOCATION_POLICY (see 16.5.6, "Location Policy Component" - CORBA 2.3).**
- **TAG_DCE_STRING_BINDING (see 16.5.1, "DCE-CIOP String Binding Component" - CORBA 2.3).**
- **TAG_DCE_BINDING_NAME (see 16.5.2, "DCE-CIOP Binding Name Component" - CORBA 2.3).**
- **TAG_DCE_NO_PIPES (see 16.5.3, "DCE-CIOP No pipes Component" - CORBA 2.3).**

5.6.4 Profile and Component Composition in IORs

The following rules augment the preceding discussion:

1. Profiles must be independent, complete, and self-contained. Their use shall not depend on information contained in another profile.
2. Any invocation uses information from exactly one profile.
3. Information used to drive multiple inter-ORB protocols may coexist within a single profile, possibly with some information (e.g., components) shared between the protocols, as specified by the specific protocols.
4. Unless otherwise specified in the definition of a particular profile, multiple profiles with the same profile tag may be included in an IOR.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

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

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

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

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

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

5.6.5 IOR Creation and Scope

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

5.6.6 Stringified Object References

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

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

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

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

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

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

(1) <oref> ::= <prefix> <hex_Octets>
(2) <prefix> ::= "IOR:"

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

(3) <hex_Octets> ::= <hex_Octet> {<hex_Octet>}*
(4) <hex_Octet>::= <hexDigit> <hexDigit>
(5) <hexDigit> ::= <digit> | <a> | <b> | <c> | <d> | <e> | <f>
(6) <digit> ::= “0” | “1” | “2” | “3” | “4” | “5” |
| “6” | “7” | “8” | “9”
(7) <a>::= “a” | “A”
(8) <b>::= “b” | “B”
(9) <c>::= “c” | “C”
(10) <d>::= “d” | “D”
(11) <e>::= “e” | “E”
(12) <f>::= “f” | “F”

The hexadecimal strings are generated by first turning an object reference into an IOR, and then encapsulating the IOR using the encoding rules of CDR, as specified in GIOP 1.0. (See 6.2, ’CDR Transfer Syntax’ for more information.) The content of the encapsulated IOR is then turned into hexadecimal digit pairs, starting with the first octet in the encapsulation and going until the end. The high four bits of each octet are encoded as a hexadecimal digit, then the low four bits.

5.6.7 Object Service Context

Emerging specifications for Object Services occasionally require service-specific context information to be passed implicitly with requests and replies. (Specifications for OMG’s Object Services are contained in *CORBAservices: Common Object Service Specifications*). The Interoperability specifications define a mechanism for identifying and passing this service-specific context information as “hidden” parameters. The specification makes the following assumptions:

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

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

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

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

```
module IOP {        // IDL

    typedef unsigned long          ServiceId;

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

    const ServiceId                TransactionService = 0;
    const ServiceId                CodeSets = 1;
    const ServiceId                ChainBypassCheck = 2;
    const ServiceId                ChainBypassInfo = 3;
    const ServiceId                LogicalThreadId = 4;
    const ServiceId                BI_DIR_IIOP = 5;
    const ServiceId                SendingContextRunTime = 6;
    const ServiceId                INVOCATION_POLICIES = 7;
    const ServiceId                FORWARDED_IDENTITY = 8;
    const ServiceId                UnknownExceptionInfo = 9;
};
```

The context data for a particular service will be encoded as specified for its service-specific OMG IDL definition, and that encoded representation will be encapsulated in the **context_data** member of **IOP::ServiceContext**. (See 6.2.2.9, 'Encapsulation'). The **context_id** member contains the service ID value identifying the service and data format. Context data is encapsulated in octet sequences to permit ORBs to handle context data without unmarshaling, and to handle unknown context data types.

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

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by the OMG. Service context ID values are of type unsigned long. Object service

specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

The **ServiceId**s currently defined are:

- **TransactionService** identifies a CDR encapsulation of the **CosTSInteroperation::PropogationContext** defined in *CORBAservices: Common Object Services Specifications*. (See 2.2, 'Other Specifications' for additional information).
- **CodeSets** identifies a CDR encapsulation of the **CONV_FRAME::CodeSetContext** defined in 5.7.2.5, 'GIOP Code Set Service Context'.
- COM-CORBA Interworking uses three service contexts as defined in "DCOM-CORBA Interworking" in the "Interoperability with non-CORBA Systems" chapter as described in CORBA 2.3 (see 2.2, 'Other Specifications' for additional information). They are:
  - **ChainBypassCheck**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassCheck**. This is carried only in a **Request** message as described in 20.9.1, "CORBA Chain Bypass" - CORBA 2.3.
  - **ChainBypassInfo**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassInfo**. This is carried only in a **Reply** message as described in 20.9.1, "CORBA Chain Bypass" - CORBA 2.3.
  - **LogicalThreadId**, which carries a CDR encapsulation of the **struct CosBridging::LogicalThreadId** as described in 20.10, "Thread Identification" - CORBA 2.3.
- **BI_DIR_IIOP** identifies a CDR encapsulation of the **IIOP::BiDirIIOPServiceContext** defined in 6.7, 'Bi-Directional GIOP'.
- **SendingContextRunTime** identifies a CDR encapsulation of the IOR of the **SendingContext::RunTime** object in 5.6, "Access to the Sending Context Run Time" - CORBA 2.3.
- **UnknownExceptionInfo** identifies a CDR encapsulation of a marshaled instance of a **java.lang.throwable** or one of its subclasses as described in Java to IDL Language Mapping.
- The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in 6.6.2, 'IIOP IOR Profiles'.
- The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures, described in 5.6, 'An Information Model for Object References'.
- The **component_data member** identifies a CDR encapsulation of a **BindingNameComponent** structure, described in 16.5.2.1, "BindingNameComponent" - CORBA 2.3.

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

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

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

- The service context is not in the OMG-defined range:
  - The receiving ORB may choose to ignore it, process it if it "understands" it, or raise a system exception, however it must be passed on through a bridge. If a system exception is raised, it shall be BAD_PARAM with an OMG standard minor code of 1.

The association of service contexts with GIOP versions, (along with some other supported features tied to GIOP minor version), is shown in Table 1.

Table 1 - GIOP Features and Versions

Feature	Version 1.0	Version 1.1	Version 1.2
Transaction Service Context	yes	yes	yes
Codeset Negotiation Service Context		yes	yes
DCOM Bridging Service Contexts: **ChainBypassCheck** **ChainBypassInfo** **LogicalThreadId**			yes
Object by Value Service Context: **SendingContextRunTime**			yes
Bi-Directional IIOP Service Context: BI_DIR_IIOP			yes
Java Language Throwable Service Context: UnknownExceptionInfo			yes
IOR components in IIOP profile		yes	yes
TAG_ORB_TYPE		yes	yes
TAG_CODE_SETS		yes	yes
TAG_ALTERNATE_IIOP_ADDRESS			yes
TAG_ASSOCIATION_OPTION		yes	yes
TAG_SEC_NAME		yes	yes
TAG_SSL_SEC_TRANS		yes	yes
TAG_GENERIC_SEC_MECH		yes	yes
TAG_*_SEC_MECH		yes	yes
TAG_JAVA_CODEBASE			yes
IOR component nn			yes
Extended IDL data types		yes	yes
Bi-Directional GIOP Features			yes

© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

5.7 Code Set Conversion

5.7.1 Character Processing Terminology

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

5.7.1.1 Character Set

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

5.7.1.2 Coded Character Set, or Code Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation or numeric value. In this specification, the term “code set” is used as an abbreviation for the term “coded character set.” Examples include ASCII, ISO 8859-1, JIS X0208 (which includes Roman characters, Japanese hiragana, Greek characters, Japanese kanji, etc.) and Unicode.

5.7.1.3 Code Set Classifications

Some language environments distinguish between byte-oriented and “wide characters.” The byte-oriented characters are encoded in one or more 8-bit bytes. A typical single-byte encoding is ISO/IEC 8859-1 (Latin-1) as used for western European languages like English. A typical multi-byte encoding which uses from one to three 8-bit bytes for each character is eucJP (Extended UNIX Code - Japan, packed format) as used for Japanese workstations.

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

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

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

5.7.1.4 Narrow and Wide Characters

Some language environments distinguish between “narrow” and “wide” characters. Typically the narrow characters are considered to be 8-bit long and are used for western European languages like English, while the wide characters

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

are 16-bit or 32-bit long and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits are insufficient. However, as noted above there are common encoding schemes in which Asian characters are encoded using multi-byte code sets and it is incorrect to assume that Asian characters are always encoded as “wide” characters.

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

5.7.1.5 Char Data and Wchar Data

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

5.7.1.6 Byte-Oriented Code Set

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

5.7.1.7 Multi-Byte Character Strings

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

5.7.1.8 Non-Byte-Oriented Code Set

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

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

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

5.7.1.10 Process Code Set and File Code Set

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

When a process needs to write international character information out to a file, or communicate with another process (possibly over a network), it typically uses a different encoding called a “file code set.” In this specification, unless otherwise indicated, all references to a program’s code set refer to the file code set, not the process code set. Even when a client and server are located physically on the same machine, it is possible for them to use different file code sets.

5.7.1.11 Native Code Set

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

5.7.1.12 Transmission Code Set

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

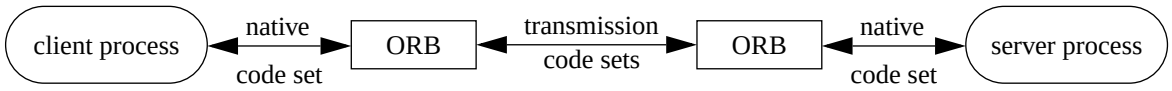


Figure 6 - Transmission Code Sets

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

5.7.1.13 Conversion Code Set (CCS)

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

5.7.2 Code Set Conversion Framework

5.7.2.1 Requirements

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

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

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

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

4. CMIR and SMIR support. The conversion framework should be flexible enough to allow conversion to be performed either on the client or server side. For example, if a client is running in a memory-constrained environment, then it is desirable for code set converters to reside in the server and for a Server Makes It

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Right (SMIR) conversion method to be used. On the other hand, if many servers are executed on one server machine, then converters should be placed in each client to reduce the load on the server machine. In this case, the conversion method used is Client Makes It Right (CMIR).

5.7.2.2 Overview of the Conversion Framework

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

The conversion framework is illustrated in Figure 7. The server-side ORB stores a server's code set information in a component of the IOR multiple-component profile structure (see 5.6.2, 'Interoperable Object References: IORs')[2]. The code sets actually used for transmission are carried in the service context field of an IOP (Inter-ORB Protocol) request header (see 5.6.7, 'Object Service Context' and 5.7.2.5, 'GIOP Code Set Service Context'). Recall that there are two code sets (TCS-C and TCS-W) negotiated for each session.

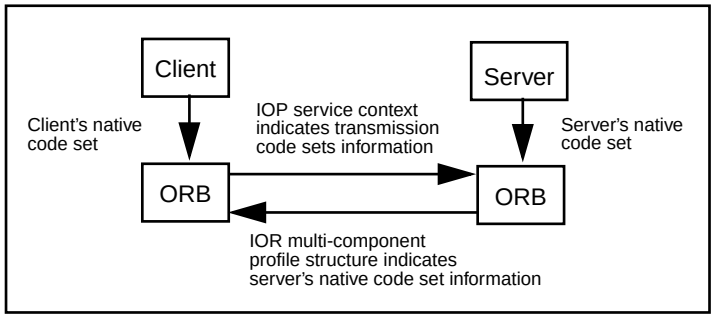


Figure 7 - Code Set Conversion Framework Overview

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

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

The rationale for selecting two transmission code sets rather than one (which is typically inferred from the locale of a process) is to allow efficient data transmission without any conversions when the client and server have identical

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

5.7.2.3 ORB Databases and Code Set Converters

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

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

5.7.2.4 CodeSet Component of IOR Multi-Component Profile

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

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

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

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

This tag has been assigned by OMG (see 5.6.3, 'Standard IOR Components'). The following IDL structure defines the representation of code set information within the component:

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

© ISO/IEC 2003 - All rights reserved

--`,,`--`,,`,,`,,`,,`---
Copyright International Organization for Standardization

```
    struct CodeSetComponentInfo {
        CodeSetComponent        ForCharData;
        CodeSetComponent        ForWcharData;
    };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (see 5.9.1, 'Character and Code Set Registry' for further information). Data within the code set component is represented as a structure of type **CodeSetComponentInfo**, and is encoded as a CDR encapsulation. In other words, the **char** code set information comes first, then the **wchar** information, represented as structures of type **CodeSetComponent**.

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

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

5.7.2.5 GIOP Code Set Service Context

The code set GIOP service context contains:

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

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

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

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

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

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (see 5.9.1, 'Character and Code Set Registry' for further information).

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

5.7.2.6 Code Set Negotiation

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

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

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

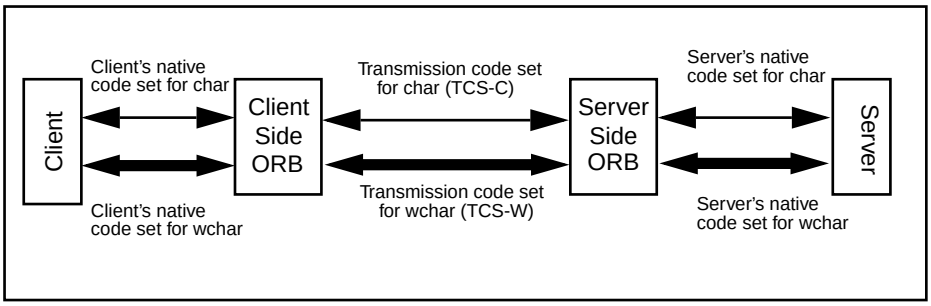


Figure 8 - Transmission Code Set Use

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

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

The client-side ORB first compares the native code sets of the client and server. If they are identical, then the transmission and native code sets are the same and no conversion is required. In this example, they are different, so code set conversion is necessary. Next, the client-side ORB checks to see if the server's native code set, eucJP, is one of the conversion code sets supported by the client. It is, so eucJP is selected as the transmission code set, with the client (i.e., its ORB) performing conversion to and from its native code set, SJIS, to eucJP. Note that the client may first have to convert all its data described as **char**s (and possibly **wchar_t**s) from process codes to SJIS first.

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

- CNCS - Client Native Code Set;
- CCCS - Client Conversion Code Sets;

© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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

The algorithm is as follows:

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

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

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

If the third option is selected and there is more than one possible intermediate conversion code set (i.e., the intersection of CCCS and SCCS contains more than one code set), then the one most preferable to the server is selected.[3]

If none of these conversions is possible, then the fallback code set (UTF-8 for **char** data and UTF-16 for **wchar** data— see below) is used. However, before selecting the fallback code set, a compatibility test is performed. This test looks at the character sets encoded by the client and server native code sets. If they are different (e.g., Korean and

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

French), then meaningful communication between the client and server is not possible and a CODESET_INCOMPATIBLE exception is raised. This test is similar to the DCE compatibility test and is intended to catch those cases where conversion from the client native code set to the fallback, and the fallback to the server native code set would result in massive data loss. (See 5.9, 'Relevant OSFM Registry Interfaces' for the relevant OSF registry interfaces that could be used for determining compatibility.)

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

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

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

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

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

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

5.7.3 Mapping to Generic Character Environments

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

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

---

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

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

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

- **wchar** must be mapped to the generic character type in a generic character environment.
- **wstring** must be mapped to a string of such generic characters in a generic character environment.
- The language binding files (i.e., stubs) generated for these generic environments must ensure that the generic type representation is converted to the appropriate code sets (i.e., CNCS on the client side and SNCS on the server side) before character data is given to the ORB runtime for transmission.

5.7.3.1 Describing Generic Interfaces

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

5.7.3.2 Interoperation

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

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

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

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

5.8 Example of Generic Environment Mapping

This section shows how **char**, **wchar**, **string**, and **wchar** can be mapped to the generic C/C++ macros of the Windows environment. This is merely to illustrate one possibility. This section is not normative and is applicable only in generic environments. See 5.7.3, 'Mapping to Generic Character Environments'.

5.8.1 Generic Mappings

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

5.8.2 Interoperation and Generic Mappings

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

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

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

5.9 Relevant OSFM Registry Interfaces

5.9.1 Character and Code Set Registry

The OSF character and code set registry is defined in *OSF Character and Code Set Registry* (see 2.2, 'Other Specifications') and current registry contents may be obtained directly from the Open Software Foundation (obtain via anonymous ftp to ftp.opengroup.org:/pub/code_set_registry). This registry contains two parts: character sets and code sets. For each listed code set, the set of character sets encoded by this code set is shown.

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

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

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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

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

5.9.2 Access Routines

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

The following routines are adopted from *RPC Runtime Support For I18N Characters - Functional Specification* (see 2.2, ’Other Specifications’).

5.9.2.1 dce_cs_loc_to_rgy

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

Synopsis

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

Parameters

Input

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

Output

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

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

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

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

The possible status codes and their meanings are as follows:

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

Description

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

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

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

5.9.2.2 dce_cs_rgy_to_loc

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

Synopsis

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

Parameters

Input

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

Output

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

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

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

The possible status codes and their meanings are as follows:

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

Description

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

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

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

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

5.9.2.3 rpc_cs_char_set_compat_check

Evaluates character set compatibility between a client and a server.

Synopsis

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

Parameters

Input

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

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Output

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

The possible status codes and their meanings are as follows:

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

Description

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

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

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

5.9.2.4 rpc_rgy_get_max_bytes

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

Synopsis

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

Parameters

Input

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

Output

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

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

The possible status codes and their meanings are as follows:

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

Description

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

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

6 General Inter-ORB Protocol

This clause 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 clause 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.

6.1 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

- ***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.

6.1.1 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 6.9, 'OMG IDL'. Fragments of the specification are used throughout this chapter as necessary.

6.1.2 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.1.3 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.

6.1.4 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 6.4, 'GIOP Message Transport'). 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, as specified in the BiDirectional GIOP specification, see 6.7, 'Bi-Directional GIOP'.
- ***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.

6.2 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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 6.2.1.1, 'Alignment'.

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 6.3, 'GIOP Message Formats'.

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.

6.2.1 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see 6.3, 'GIOP Message Formats') 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 6.2.2.9, 'Encapsulation'. 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.

6.2.1.1 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 2 - gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 2 - Alignment Boundaries

TYPE	OCTET ALIGNMENT
char	1
wchar	1, 2, or 4, depending on code set
octet	1
short	2
unsigned short	2
long	4
unsigned long	4

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Table 2 - Alignment Boundaries

TYPE	OCTET ALIGNMENT
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 6.3.1, 'GIOP Message Header') and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See 6.2.2.9, 'Encapsulation').

6.2.1.2 Integer Data Types

Figure 9 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.

6.2.1.3 Floating Point Data Types

Figure 10 illustrates the representation of floating point numbers. These exactly follow the IEEE standard formats for floating point numbers[4], 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.

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:

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

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

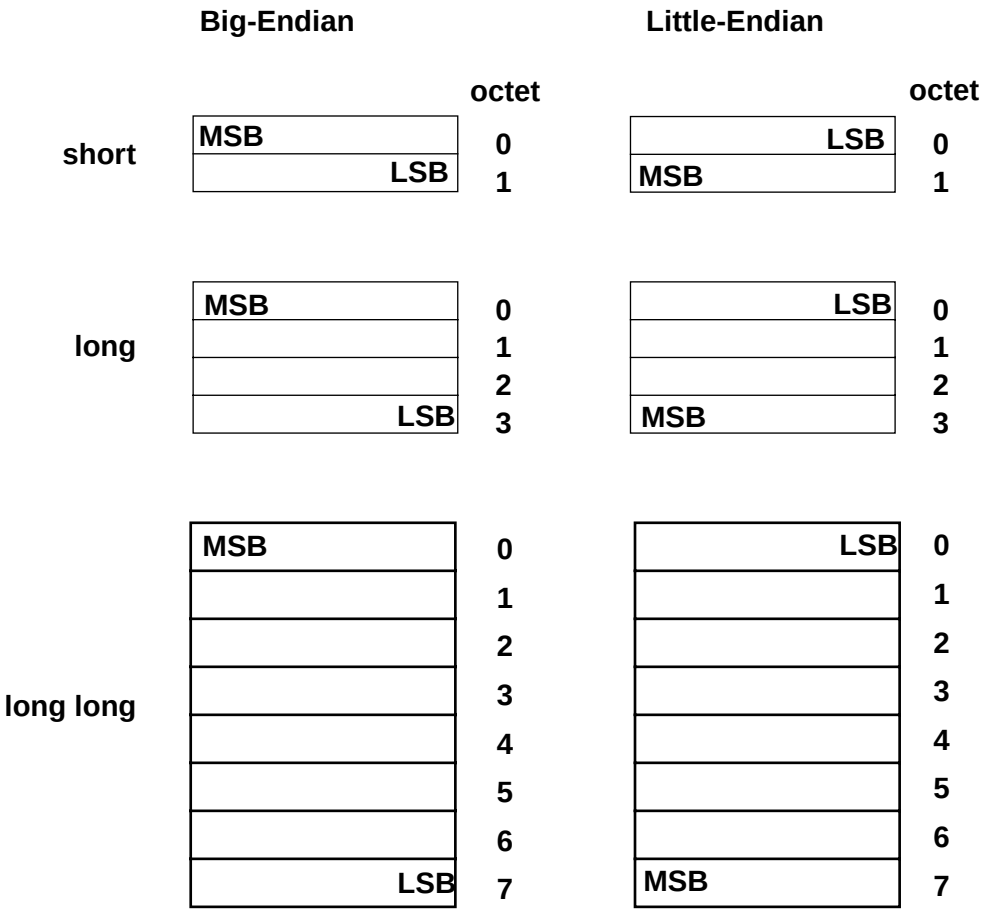


Figure 9 - Sizes and bit ordering in big-endian and little-endian encodings of OMG IDL integer data types, both signed and unsigned.

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:

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

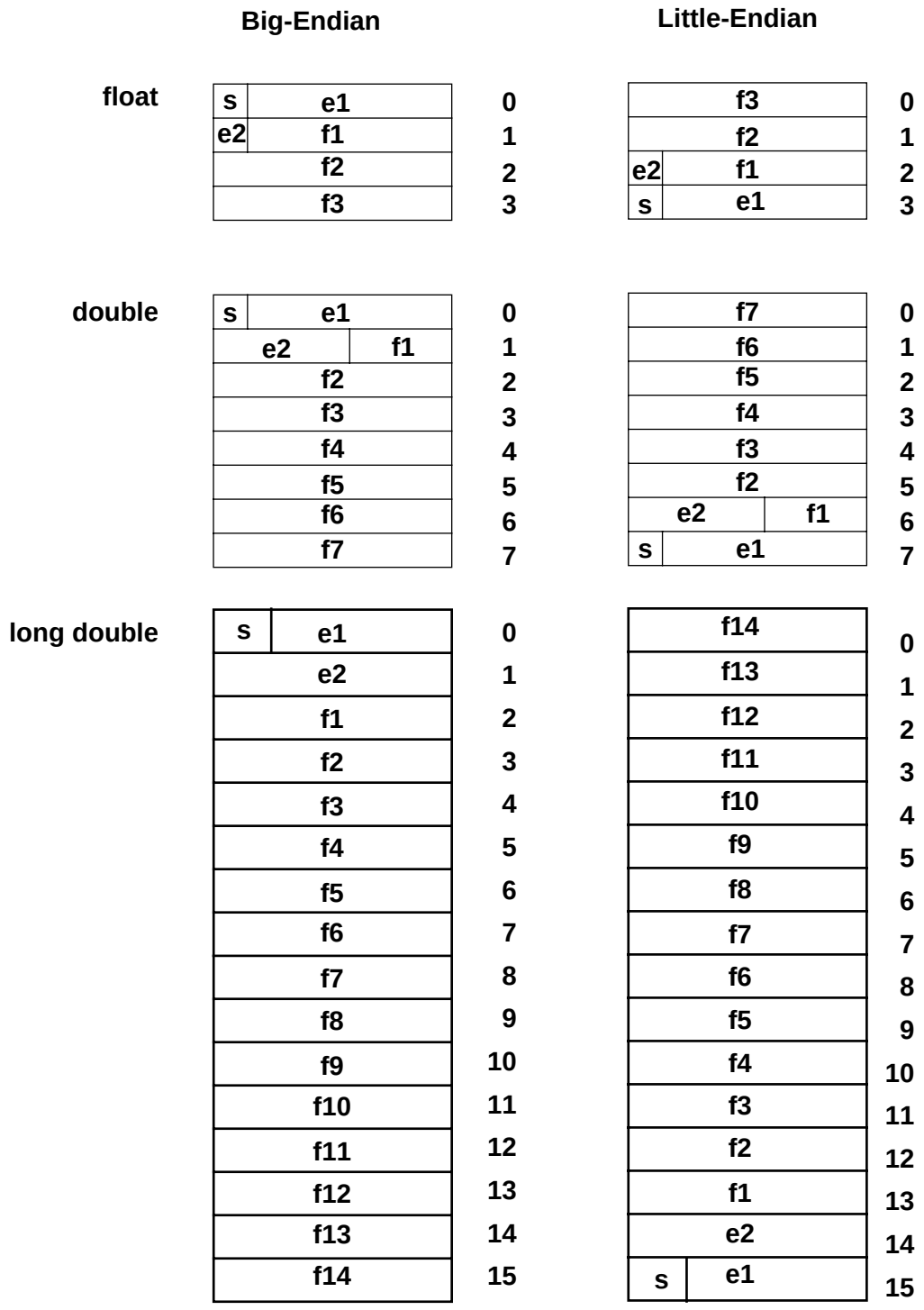


Figure 10 - Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single, double precision, and double extended floating point numbers.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.2.1.4 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.

6.2.1.5 Boolean

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

6.2.1.6 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 5.7, 'Code Set Conversion'. 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 5.7, 'Code Set Conversion') 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 5.9, 'Relevant OSFM Registry Interfaces' 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, **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 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, 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 CDR.

6.2.2 OMG IDL Constructed Types

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

6.2.2.1 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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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.

6.2.2.2 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.

6.2.2.3 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.

6.2.2.4 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.

6.2.2.5 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.

6.2.2.6 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.

6.2.2.7 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. Both the string length and contents include a terminating null.

For GIOP version 1.1 and 1.2, 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. Both the string length and contents include a terminating null. The terminating null character for a **wstring** is also a wide character.

For GIOP version 1.2, 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 a **wstring** is not terminated by a **NUL** character. In particular, in GIOP version 1.2 a length of 0 is legal for **wstring**.

Note – For GIOP versions prior to 1.2, 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 CDR.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.2.2.8 Fixed-Point Decimal Type

The IDL **fixed** type has no alignment restrictions, and is represented as shown in Figure 12 -. 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.

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 11 - Decimal Digit Encoding for Fixed Type

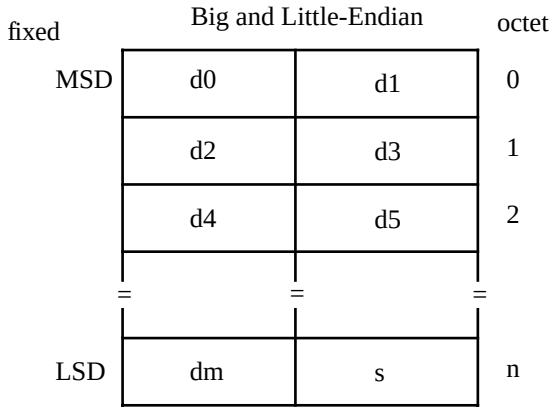


Figure 12 - IDL Fixed Type Representation

6.2.2.9 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 6.2.4.1, 'TypeCode'), the IIOP protocol profile inside an IOR (see 6.2.5, 'Object References'), and in service-specific context (see 5.6.7, 'Object Service Context'). In addition, some ORBs may choose to use an encapsulation to hold the **object_key** (see 6.6.2, 'IIOP IOR Profiles'), or in other places that a **sequence<octet>** data type is in use.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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 6.3.1, 'GIOP Message Header'). 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 6.2.2.5, 'Sequence'). 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.[5]

6.2.3 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.

6.2.3.1 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.

5. 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. The GIOP protocol itself does not require encapsulation of such data.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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 6.2.6, 'Abstract Interfaces'), **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.

6.2.3.2 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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 in the 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 in the 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 in the 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)

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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

valuetype horse: animal ...

then the encoding is as shown in the 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 in the 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.

6.2.3.3 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 which has already been encoded, a **RepositoryID** which has already been encoded, a list of repository IDs which has already been encoded, or another value object which is shared in a graph. **0xffffffff** is always followed by a **long** indicating where to go in the buffer.

The encoding used for 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.

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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.2.3.4 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. 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 which 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 which 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 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**.

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

```
// IDL
valuetype simple Node { ... };
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.

- 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. For the purposes of chunking, values encoded as indirections or null are treated as non-value data.

- 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 6.2.3.1, 'Partial Type Information and Versioning').

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.

6.2.3.5 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.2.3.6 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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

6.2.4 Pseudo-Object Types

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

6.2.4.1 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 3 -. 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 3 -. A “simple” parameter list has a fixed number of fixed length entries, or a single parameter which 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 6.2.2.9 ”Encapsulation” on page 51) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 3 -.

The third column of Table 3 - 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’. This indirection does not need to be exposed to application programmers.

TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 3 -. The ordering and meaning of parameters is a superset of those given in 10.7, “TypeCodes” - CORBA 2.3. 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.[6]

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Table 3 - TypeCodes

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*[a] (label value), string (member name), TypeCode (member type)}

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Table 3 - TypeCodes

TCKind	Integer Value	Type	Parameters
tk_enum	17	complex	string (repository ID), string (name), ulong (count) {string (member name)}
tk_string	18	simple	ulong (max length[b])
tk_sequence	19	complex	TypeCode (element type), ulong (max length[c])
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)[d], ulong (count), {string (member name), TypeCode (member type), short(Visibility)}

6. 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.

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

Table 3 - TypeCodes

TCKind	Integer Value	Type	Parameters
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)
– none –	0xffffffff	simple	long (indirection[e])

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

b. For unbounded strings, this value is zero.

c. For unbounded sequences, this value is zero.

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

e. See "Indirection: Recursive and Repeated TypeCodes" on page 63.

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 mandatory. 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.

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.

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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 and 1.2 introduces the possibility of a header for a **Fragment Message** 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.)

6.2.4.2 Any

Any values are encoded as a **TypeCode** (encoded as described above) followed by the encoded value.

6.2.4.3 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.

6.2.4.4 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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.

6.2.5 Object References

Object references are encoded in OMG IDL (as described in 5.5, 'Object Addressing'). 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 6.6.2, 'IIOP IOR Profiles').

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.

6.2.6 Abstract Interfaces

Abstract interfaces are encoded as a union with a **boolean** discriminator. The **union** has an *object reference* (see 6.2.5, 'Object References') if the discriminator is **TRUE**, and a *value type* (see 6.2.3, 'Value Types') 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.

6.3 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 6.4.1, 'Connection Management') 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, either side may originate messages, as specified in 6.7, 'Bi-Directional GIOP'.

GIOP message types are summarized in the table below, 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.

Message Type	Originator	Value	GIOP Versions
Request	Client	0	1.0, 1.1, 1.2
Reply	Server	1	1.0, 1.1, 1.2
CancelRequest	Client	2	1.0, 1.1, 1.2
LocateRequest	Client	3	1.0, 1.1, 1.2

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

Message Type	Originator	Value	GIOP Versions
LocateReply	Server	4	1.0, 1.1, 1.2
CloseConnection	Server	5	1.0, 1.1, 1.2
MessageError	Both	6	1.0, 1.1, 1.2
Fragment	Both	7	1.1, 1.2

6.3.1 GIOP Message Header

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

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

© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

```
    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
    typedef MessageHeader_1_1 MessageHeader_1_2;
};
```

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 (ISO/IEC 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) which constitute the GIOP. This is not equivalent to the IIOP version number (as described in 6.2.5, ’Object References’) 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

- **flags** (in GIOP 1.1 and 1.2) 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 fragments 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 and 1.2.

- **message_type** indicates the type of the message, 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. 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, 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.

6.3.2 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

6.3.2.1 Request Header

The request header is specified as follows:

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

    // 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;
      Principal                     requesting_principal;
    };
```

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

```
// 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;
   Principal                        requesting_principal;
};

// GIOP 1.2
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;
```


© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

```
        // Principal not in GIOP 1.2
    };
};
```

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 6.3.4, 'CancelRequest Message').

- The lowest order bit of **response_flags** is set to **1** if a reply message is expected for this request. If the operation is not defined as **oneway**, and the request is not invoked via the DII with the **INV_NO_RESPONSE** flag set, **response_flags** must be set to **\x03**.

  If the operation is defined as oneway, or the request is invoked via the DII with the **INV_NO_RESPONSE** flag set, **response_flags** may be set to **\x00 or \x01**. Asking for a reply gives the client ORB an opportunity to receive **LOCATION_FORWARD** responses and replies that might indicate system exceptions. When this flag is set to **\x01** for a oneway operation, receipt of a reply does not imply that the operation has necessarily completed.

- **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 which 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, **target** identifies the object which 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.

- **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 "Object Reference Operations" - CORBA 2.3 and that correspond to GIOP request messages, the operation names are **_interface**, **_is_a**, **_non_existent,** and **_get_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**.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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 5.6.7, 'Object Service Context'.
- **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.

6.3.2.2 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, 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 6.2.4.4, 'Context'. 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
};
```

6.3.3 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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.3.3.1 Reply Header

The reply header is defined as follows:

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

#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
    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
    };
```

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

```
    struct ReplyHeader_1_2 {
      unsigned long                 request_id;
      ReplyStatusType_1_2           reply_status;
      IOP:ServiceContextList        service_context;
    };
#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 6.1.4, 'GIOP Message Transfer'.

6.3.3.2 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, 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 6.2.4.4, 'Exception'. (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;
```


© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

```
        };
};
```

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 define 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** in 3.17.2, “Standard Minor Exception Codes” - CORBA 2.3 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 6.2.5, ’Object References’. The client ORB is responsible for resending 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.

For example, the reply body for a successful response (the value of **reply_status** is **NO_EXCEPTION**) to the Request example shown previously 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 6.5, ’Object Location’.

6.3.4 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:

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

6.3.4.1 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.

6.3.5 LocateRequest Message

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

- whether the object reference is valid,
- 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**

6.3.5.1 LocateRequest Header

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

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

  // GIOP 1.0
    struct LocateRequestHeader_1_0 {
    // Renamed LocationRequestHeader
```


© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

```
        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
    struct LocateRequestHeader_1_2 {
        unsigned long          request_id;
        TargetAddress          target;
    };
};
```

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 6.3.2, 'Request Message' 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, 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 6.5, 'Object Location' for details on the use of **LocateRequest.**

6.3.6 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**

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

3. The locate reply body

6.3.6.1 Locate Reply Header

The locate reply header is defined as follows:

```
module GIOP {                    // IDL extended for GIOP 1.2
#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
    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 {
```


© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

```
        unsigned long               request_id;
        LocateStatusType_1_2        locate_status;
    };
#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.

6.3.6.2 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.

6.3.7 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 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 6.4.1, 'Connection Management'.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

6.3.8 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.

6.3.9 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, 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 which 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.

For GIOP version 1.2, 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, 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.

```
module GIOP {//IDL extension for GIOP 1.2
        // GIOP 1.2
        struct FragmentHeader_1_2 {
                unsigned long request_id;
        };
};
```


© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization

6.4 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 6.4.1, 'Connection Management' 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.

6.4.1 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 the client can send them as well. Either client or server can send **MessageError** messages, in GIOP 1.0 and 1.1.

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.

6.4.1.1 Connection Closure

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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:

- 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 which 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 6.7, 'Bi-Directional GIOP' apply.

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

- 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 resent 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.

6.4.1.2 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 re-using 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.

6.4.2 Message Ordering

Only the client (connection originator) may send **Request, LocateRequest, and CancelRequest** messages. Connections are not fully symmetrical.

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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.

6.5 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, 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

© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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.

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.

6.6 Internet Inter-ORB Protocol (IIOP)

The baseline transport specified for GIOP is TCP/IP[7]. 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.

6.6.1 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 6.6.2, 'IIOP IOR Profiles'. 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, 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 which 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.

---

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

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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.

6.6.2 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 and 1.2
    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
        Version             iiop_version;
        string              host;
        unsigned short      port;
        sequence <octet>    object_key;

    // Added in 1.1 unchanged for 1.2
        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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

- 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 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 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 1, 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. 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.

  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.

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

- **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 6.6.3, 'IIOP IOR Profile Components'. 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 presence, which are optional presence, and which can be dropped. 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 which 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.

Also note that at this time no "well-known" port number has been allocated; therefore, individual agents will need to assign previously unused ports as part of their installation procedures. IIOP supports such multiple agents per host.

6.6.3 IIOP IOR Profile Components

The following components are part of the IIOP 1.1 and 1.2 conformance. All these components are optional presence in the IIOP profile and cannot be dropped from an IIOP 1.1 or 1.2 IOR.

- **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_CSI_ECMA_Public_SEC_MECH**
- **TAG_JAVA_CODEBASE**

The following components are part of the IIOP 1.2 conformance. All these components are optional presence in the IIOP profile and cannot be dropped from an IIOP 1.2 IOR.

- **TAG_ALTERNATE_IIOP_ADDRESS**
- **TAG_POLICIES**

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

- **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**

6.7 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 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 exported by the client to the server via this connection will be sent back to the client on this same connection.

The mechanism by which the client and server agree on this capability is as follows:

The client creates an object for exporting to a server.

The client exports the IOR as a parameter of a GIOP Request on the server object. If the 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 a new POA policy has been defined (6.8, ’Bi-directional GIOP policy’).

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 6.7.1, ’Bi-Directional IIOP’.

The server receives the **Request**. If it recognizes the service context and supports bi-directional connections, it may send invocations on this object back along the connection.

The server may not wish to support bi-directionality either due to lack of support for it, or because it has been configured that way. In this case, it may fall back to initiating a connection to the object in the usual way.

© ISO/IEC 2003 - All rights reserved

--`,,`--`,,`,`,,`,`,,`---

Copyright International Organization for Standardization

If a GIOP connection is used bi-directionally, the client should attempt to keep the connection alive as long as is necessary to complete its object's service to the server. If the client initiates a new connection it is not foreseen here that the server can use that connection for requests on the object exported previously.

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 GIOP 1.1). Similarly the client can use message types traditionally used only by servers: **Reply**, **LocateReply**, **MessageError**, **CloseConnection,** and **Fragment**.

CloseConnection messages are a special case however. Either ORB may send a **CloseConnection** message, but the conditions in 6.4.1, 'Connection Management' apply.

Bi-directional GIOP connections modify the behavior of Request IDs. In the GIOP specification, 6.4.1, 'Connection Management', it is noted that "Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection". With bi-directional IIOP, the Request ID unambiguously associates replies with requests per connection *and per direction*, so the same Request ID can be used for a **Request** going from client-to-server and for a **Request** going from server-to-client, simultaneously.

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.

It is possible for a client to masquerade for a callback object, by pretending that a callback object can be reached over an existing connection to the client. If the server has doubts in the integrity of the client, it is recommended that bi-directional GIOP not be used.

6.7.1 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 {
```

© ISO/IEC 2003 - All rights reserved

--`,,`--`,,`,,`,`,,`---

Copyright International Organization for Standardization

```
        ListenPointList listen_points;
    };
};
```

The data encapsulated in the **BiDirIIOPServiceContext** structure which is identified by the ServiceId **BI_DIR_IIOP** as defined in 5.6.7, 'Object Service Context', 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 6.6.2, 'IIOP IOR Profiles'. 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. Typically, when the same client has multiple connections to the same server, the **listen_points** will be identical. However, if they differ, they supplement each other (i.e., any of the listen points received on any of the connections may be used).

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.

If a client has not set up any mechanism for traditional-style callbacks using a listening socket, then the **port** entry in its IOR must be set to the outgoing connection's local port (as retrieved using the getsockname() sockets API call). The **port** in the **BI_DIR_IIOP** structure must match this value. This will allow multiple clients, all running in restrictive security modes (such as Java applets) on the same host, all of them connecting to one server, to each receive callbacks on their correct connection.

6.7.1.1 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).

6.8 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.

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

```
// 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.

6.9 OMG IDL

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

6.9.1 GIOP Module

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

    struct Version {
      octet             major;
      octet             minor;
    };

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

```
        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
    typedef MessageHeader_1_1 MessageHeader_1_2;
```


© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

// GIOP 1.2
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;
```

© ISO/IEC 2003 - All rights reserved

--`,,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

```
        case ReferenceAddr:             IORAddressingInfo ior;
    };

    struct RequestHeader_1_2 {
        unsigned long                   request_id;
        octet                           response_flags;
        octet                           reserved[3];
        TargetAddress                   target;
        string                          operation;
        // Principal not in GIOP 1.2
        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
    enum ReplyStatusType_1_2 {
```


© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

```
        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
    };

#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
```

© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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

#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
      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
      };
```


© ISO/IEC 2003 - All rights reserved

--`,,`-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization

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

        // GIOP 1.2
        struct FragmentHeader_1_2 {
            unsigned long                   request_id;
        };
};
```

6.9.2 IIOP Module

```
        module IIOP {     // IDL extended for version 1.1 and 1.2
        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
            Version             iiop_version;
            string              host;
            unsigned short      port;
            sequence <octet>    object_key;

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

© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

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

        typedef sequence<ListenPoint> ListenPointList;

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

6.9.3 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;
        };
};
```


© ISO/IEC 2003 - All rights reserved

Copyright International Organization for Standardization

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 35.080

Price based on 96 pages

--`,,`-`-`,,`,,`,`,,`---

© ISO/IEC 2003 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale