CORBA for embedded Specification

This OMG document replaces the submission document (realtime/06-02-02) and the draft adopted specification (ptc/06-05-01). It is an OMG Final Adopted Specification and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by August 25, 2006.

You may view the pending issues for this specification from the OMG revision issues web page *http://cgi.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on April 6, 2007. If you are reading this after that date, please download the available specification from the OMG formal specifications web page.

NOTE: This document supersedes ptc/06-07-02

OMG Adopted Specification
ptc/06-08-03

Date: August 2006

Common Object Request Broker Architecture (CORBA) *for embedded* Specification

This is the core CORBA/*e* specification. Mappings to particular programming languages, and the Common Object Services are documented in the other specifications that together comprise the complete CORBA specification. Please visit the CORBA download page at *http://www.omg.org/technology/documents/corba_spec_catalog.htm* to find the complete CORBA specification set.

ptc/06-08-03

NOTE: This document supersedes ptc/06-07-02.

OBJECT MANAGEMENT GROUP

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

Context of CORBA

The key to understanding the structure of the CORBA architecture is the Reference Model, which consists of the following components:

- **Object Request Broker** - enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.
- **Object Services** - a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices: Common Object Services Specification*.
- **Common Facilities** - a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities will be contained in *CORBAfacilities: Common Facilities Architecture*.
- **Application Objects** - products of a single vendor or in-house development group that controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. Combined with the Object Services, it ensures meaningful communication between CORBA-compliant applications.

CORBA Interoperability

The key to CORBA's success is its interoperability, an attribute that extends the utility of CORBA across an immense range of computing hardware infrastructures, operating systems, and computing languages while allowing very different CORBA implementations suited to the local operating environment. Thus, CORBA is available for non-stop fault-tolerant super-computers, enterprise tier servers, personal computers, and embedded systems such as hand-held radios. Applications may be developed that benefit from all these computing regimes because the differently-targeted implementation of CORBA are all interoperable.

Context of CORBA/e

Prior to 2005, CORBA Specifications were organized in a somewhat piece meal fashion that reflected their creation over an extended period of time. Since CORBA 1.1, in 1991, each area of further standardization was either added as a new chapter to the CORBA "book" or became a separate specification altogether.

This made the picture around conformance confused. Each chapter contained optional conformance points, conformance points that spanned chapters were oblique and hard to locate. Between specifications that remained outside of the CORBA book, there was no conformance relationship at all.

It also led to a slow deterioration in architectural coherence of CORBA, due to the misplacement of new prescriptions. This was prone to inconsistencies, both in the mind of the reader and also across the specification itself.

The new packagings of the CORBA specifications address these problems. Optional conformance points are resolved. Redundant prescriptions are removed, where this does not break the CORBA typing rules. The document has been re-structured to reflect the architecture more faithfully.

The net result of these editorial changes is a document that is focused at a major use-case for CORBA. That is, solving the problems inherent with integrating embedded distributed systems. With this in mind, this specification is the "CORBA/e" specification, which can be read as “CORBA for embedded.” CORBA/e is the OMG’s umbrella term representing a family of specifications: the CORBA/e Profiles. CORBA/e is targeted at developers of Distributed Real-time Embedded (DRE) systems.

CORBA/e enables the implementation of middleware products that are open, mature, robust, and memory efficient (small footprint); offering high performance and deterministic real-time behavior - key requirements of most DRE systems. CORBA/e consolidates CORBA as the dominant standard for embedded middleware for the foreseeable future.

It is further recognized that there are a wide range of capabilities in today’s embedded systems. A key tenet of CORBA/e is the support for “Profiling,” a process by which specific subsets of these CORBA specifications can be defined by different end user communities and then standardized through the OMG. This will allow different end user communities to choose the appropriate subset of CORBA features required by any standards that are developed within that working group/task force and then for the vendors to produce highly optimized implementations based on the corresponding CORBA/e specification (Profile).

Thus, two profiles of CORBA/*e* are defined. Middleware vendors are able to claim CORBA/e compliance by implementing one or more of the CORBA/e Profiles:

- **Compact Profile** - The CORBA/*e* Compact Profile was formerly known as the Minimum CORBA specification is targeted to applications or portions of applications that will be executing on an embedded processor with constrained resources but at the same time requiring predictable real-time behavior. Assumed resources include those that are typically available in a 32-bit micro-processor running any one of the available Real-Time Operating Systems (RTOSes). Assumed application requirements are limited to statically defined interfaces, interactions, and scheduling. While resource constrained, these systems typically have enough capability to support sophisticated embedded applications, such as signal or image processing, with real-time predictability requirements.

- **Micro Profile** - The CORBA/*e* Micro Profile is targeted to applications executing on a “deeply embedded” processor with severely constrained resources, e.g., those in a mobile device. Assumed resources are limited to those that are typically available on a low-power microprocessor or high-end Digital Signal Processor (DSP). Assumed application requirements are limited to statically defined interfaces, interactions, and scheduling.

The profiles presented in this book are a strict profile of previous CORBA specifications.

Applications that use implementations of these profiles retain interoperability, both between implementations that conform to the different profiles and with CORBA implementations, such as those that conform to the CORBA/*i* (CORBA *for integration*) specification.

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications

- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary, if applicable. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

This specification is targeted to applications or portions of applications that will be executing on an embedded processor with constrained resources and/or that requires predictable real-time behavior.

1.1 Intended Audience

The architecture and specifications described in this manual are aimed at software designers and developers of Distributed Real-Time Embedded (DRE) Systems who want to produce embedded applications that comply with OMG standards for the Object Request Broker (ORB). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. The ORB provides the mechanisms by which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

2 Conformance and Compliance

2.1 Definition of CORBA/e Compliance

Compliance to CORBA/e means conformance with one or more of the profiles presented in this specification.

As with all OMG standards, this specification is inclusive, rather than proscriptive; it sets the minimum requirements for a compliant implementation. Implementations may offer extensions, including extensions defined within other OMG specifications, without affecting compliance.

Compliance to the CORBA/e Compact Profile means conformance with every prescription for the CORBA/e Compact Profile contained within the conformance section of all chapters and conformance to at least one OMG-standardized mapping of IDL to a programming language.

Compliance to the CORBA/e Micro Profile means conformance with every prescription for the CORBA/e Micro Profile contained within the conformance section of all chapters and conformance to at least one OMG-standardized mapping of IDL to a programming language.

2.2 Definition of Minimum CORBA Compliance

The CORBA/e Compact Profile replaces the Minimum CORBA standard. All statements of compliance to the Minimum CORBA standard, unless specifically stated to be compliance to Minimum CORBA 1.0, are equivalent to and must meet the requirements of compliance with the CORBA/e Compact Profile.

3 Additional Information

3.1 Structure of Manual

This manual is divided into the four parts of Object Model, Interchange Formats, Interfaces, and Interoperability. The specification contains the following chapters:

Object Model

Chapter 1- CORBA Overview contains the overall structure of the ORB architecture and includes information about CORBA interfaces and implementations.

Chapter 2- The Object Model describes the computation model that underlies the CORBA architecture.

Interchange Formats

Chapter 3 - OMG IDL Syntax and Semantics details the OMG interface definition language (OMG IDL), which is the language used to describe the interfaces that client objects call and object implementations provide.

Chapter 4 - Repository IDs defines the string-based IDs that are generated to identify types defined in IDL and the elements of IDL that control them.

Interfaces

Chapter 5- ORB Interface defines the interface to the ORB functions that do not depend on object adapters: these operations are the same for all ORBs and object implementations.

Chapter 6 - Object Interfaces explains the operations and other facilities provided for the object-oriented constructs defined in OMG IDL, including interfaces and value types.

Chapter 7 - Policy explains the use of policy objects that can be used to modify the behavior of the ORB or of objects.

Chapter 8 - Portable Object Adapter defines a group of interfaces that an implementation uses to access ORB functions.

Chapter 9 - Real-time defines the architecture and real-time features for CORBA/*e*.

Chapter 10 - Naming Service defines the standard service of CORBA/*e* that provides for association of the names with object references, e.g., for bootstrapping systems.

Chapter 11 - Event Service defines the standard service of CORBA/*e* that provides for asynchronous multi-point to multi-point delivery of information.

Chapter 12 - Lightweight Logging Service defines the standard service of CORBA/*e* that provides for maintenance of in-memory logs of information.

Interoperability

Chapter 13 - General Inter-ORB Protocol describes the general inter-ORB protocol (GIOP) and includes information about the GIOP's goals, transport, and object location.

Chapter 14 - CDR Transfer Syntax describes the format in which the GIOP represents OMG IDL data types.

Chapter 15 - GIOP Messages describes the messages used by GIOP.

Chapter 16 - Internet Interoperability Protocol describes the use of GIOP over TCP/IP.

3.2 Acknowledgements

The following companies submitted and/or supported parts of the specifications that were approved by the Object Management Group to become *CORBA*:

- Adiron, LLC
- Alcatel
- BEA Systems, Inc.
- BNR Europe Ltd.
- Borland International, Inc.
- Compaq Computer Corporation
- Concept Five Technologies
- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- Defense Information Systems Agency
- Digital Equipment Corporation
- Ericsson
- Eternal Systems, Inc.
- Expersoft Corporation
- France Telecom
- FUJITSU LIMITED
- Genesis Development Corporation
- Gensym Corporation
- Hewlett-Packard Company
- HighComm
- Highlander Communications, L.C.
- Humboldt-University
- HyperDesk Corporation
- ICL, Plc.
- Inprise Corporation
- International Business Machines Corporation
- International Computers, Inc.
- IONA Technologies, Plc.
- Lockheed Martin Federal Systems, Inc.
- Lucent Technologies, Inc.
- Micro Focus Limited
- MITRE Corporation
- Motorola, Inc.
- NCR Corporation
- NEC Corporation
- Netscape Communications Corporation

- Nortel Networks
- Northern Telecom Corporation
- Novell, Inc.
- Object Design, Inc.
- Objective Interface Systems, Inc.
- Object-Oriented Concepts, Inc.
- OC Systems, Inc.
- Open Group - Open Software Foundation
- Oracle Corporation
- PeerLogic, Inc.
- Persistence Software, Inc.
- Promia, Inc.
- Siemens Nixdorf Informationssysteme AG
- SPAWAR Systems Center
- Sun Microsystems, Inc.
- SunSoft, Inc.
- Sybase, Inc.
- Telefónica Investigación y Desarrollo S.A. Unipersonal
- TIBCO, Inc.
- Tivoli Systems, Inc.
- Tri-Pacific Software, Inc.
- University of California, Santa Barbara
- University of Rhode Island
- Visual Edge Software, Ltd.
- Washington University

4 CORBA Overview

This chapter is non-normative introductory material. Furthermore, it describes a more general model of CORBA than is embraced by the CORBA/*e* profiles. In particular, the dynamic features of CORBA are not generally useful in embedded systems and are not included in the CORBA/*e* profiles.

4.1 Overview

The Common Object Request Broker Architecture (CORBA) is structured to allow integration of a wide variety of object systems. The motivation for some of the features may not be apparent at first, but as we discuss the range of implementations, policies, optimizations, and usages we expect to encompass, the value of the flexibility becomes more clear.

4.2 Structure of an Object Request Broker

Figure 4.1 shows a request being sent by a client to an object implementation. The Client is the entity that wishes to perform an operation on the object and the Object Implementation is the code and data that actually implements the object.

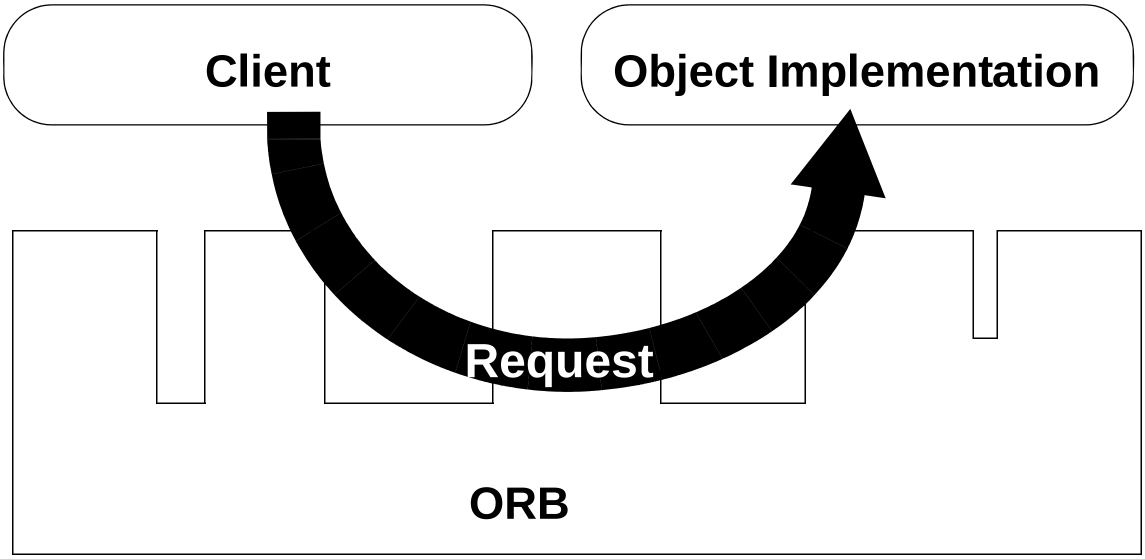

Figure 4.1- A Request Being Sent Through the Object Request Broker

The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface.

Figure 4.2 shows the structure of an individual Object Request Broker (ORB). The interfaces to the ORB are shown by striped boxes, and the arrows indicate whether the ORB is called or performs an up-call across the interface.

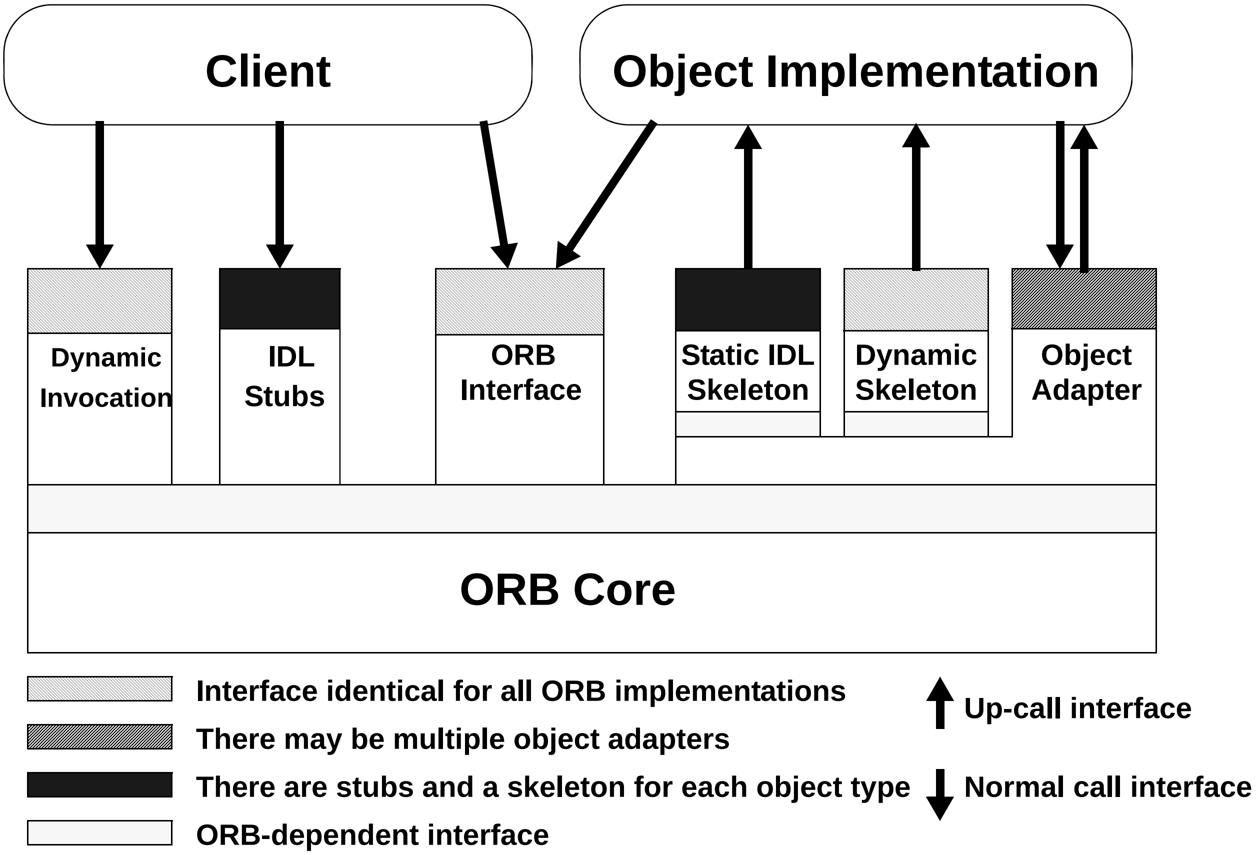

Figure 4.2- The Structure of Object Request Interfaces

To make a request, the Client can use the Dynamic Invocation interface (the same interface independent of the target object’s interface) or an OMG IDL stub (the specific stub depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.

The Object Implementation receives a request as an up-call either through the OMG IDL generated skeleton or through a dynamic skeleton. The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times.

Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language, called the OMG Interface Definition Language (OMG IDL). This language defines the types of objects according to the operations that may be performed on them and the parameters to those operations. Alternatively, or in addition, interfaces can be added to an Interface Repository service; this service represents the components of an interface as objects, permitting run-time access to these components. In any ORB implementation, the Interface Definition Language (which may be extended beyond its definition in this document) and the Interface Repository have equivalent expressive power.

The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. The client initiates the request by calling stub routines that are specific to the object or by constructing the request dynamically (see Figure 4.3).

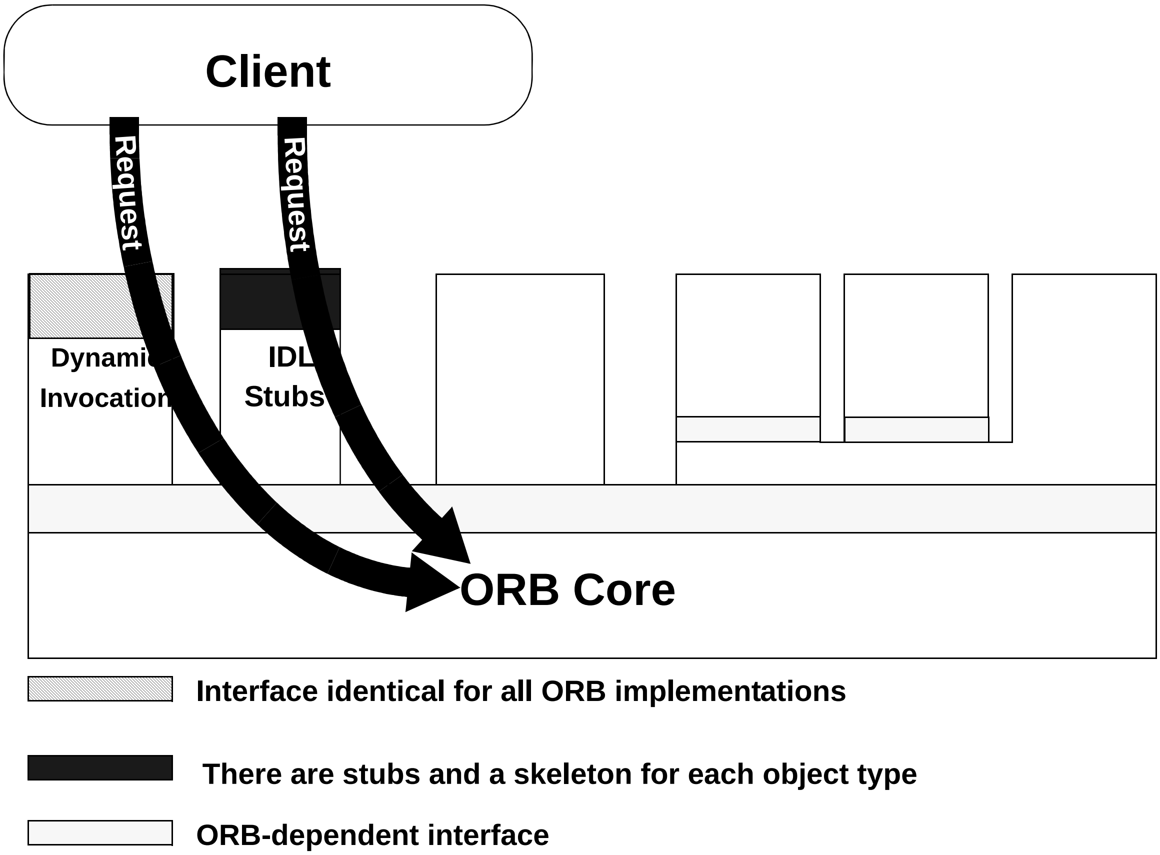
Client
Request
Request
Dynamic
Invocation
IDL
Stubs
ORB Core
Interface identical for all ORB implementations
There are stubs and a skeleton for each object type
ORB-dependent interface

Figure 4.3- A Client Using the Stub or Dynamic Invocation Interface

The dynamic and stub interface for invoking a request satisfy the same request semantics, and the receiver of the message cannot tell how the request was invoked.

The ORB locates the appropriate implementation code, transmits parameters, and transfers control to the Object Implementation through an IDL skeleton or a dynamic skeleton (see Figure 4.4). Skeletons are specific to the interface and the object adapter. In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client.

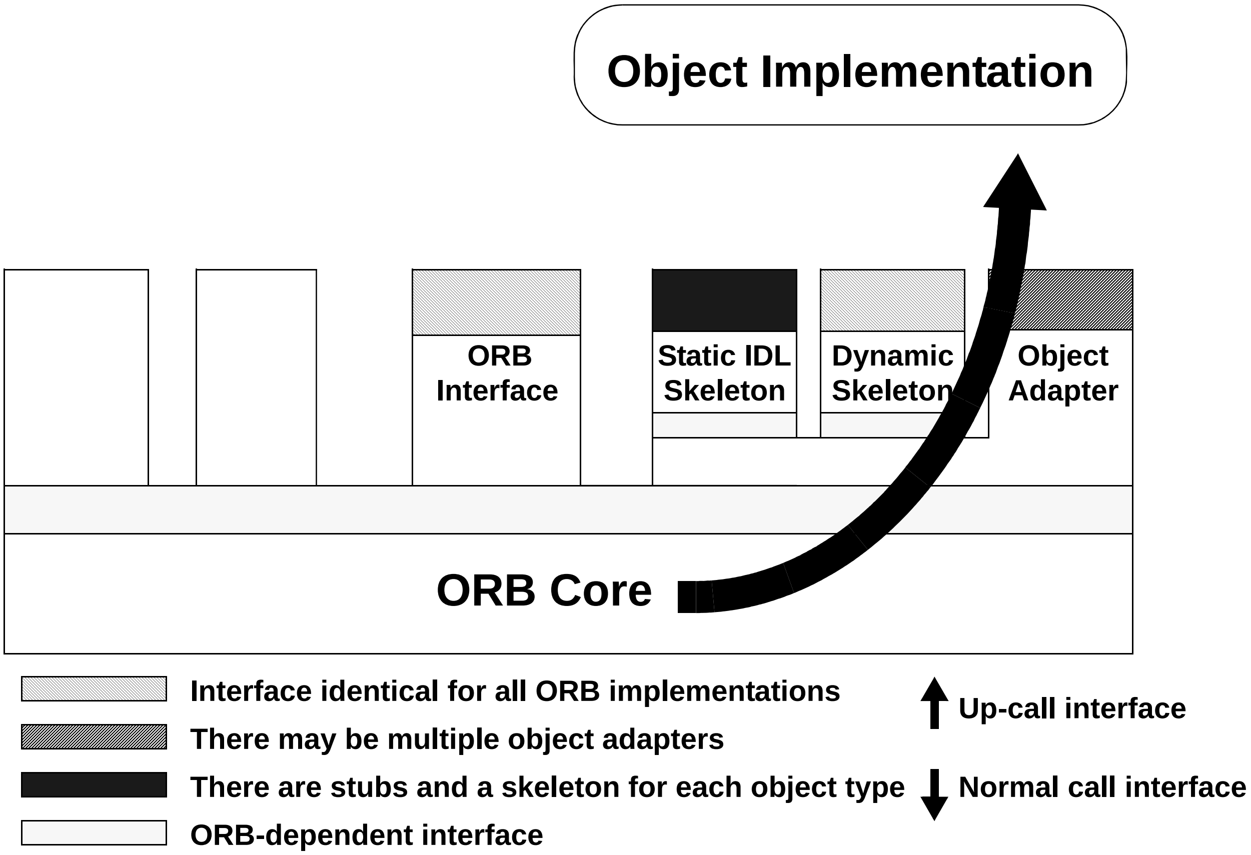

Figure 4.4- An Object Implementation Receiving a Request

The Object Implementation may choose which Object Adapter to use. This decision is based on what kind of services the Object Implementation requires.

Figure 4.5 shows how interface and implementation information is made available to clients and object implementations. The interface is defined in OMG IDL and/or in the Interface Repository; the definition is used to generate the client Stubs and the object implementation Skeletons.

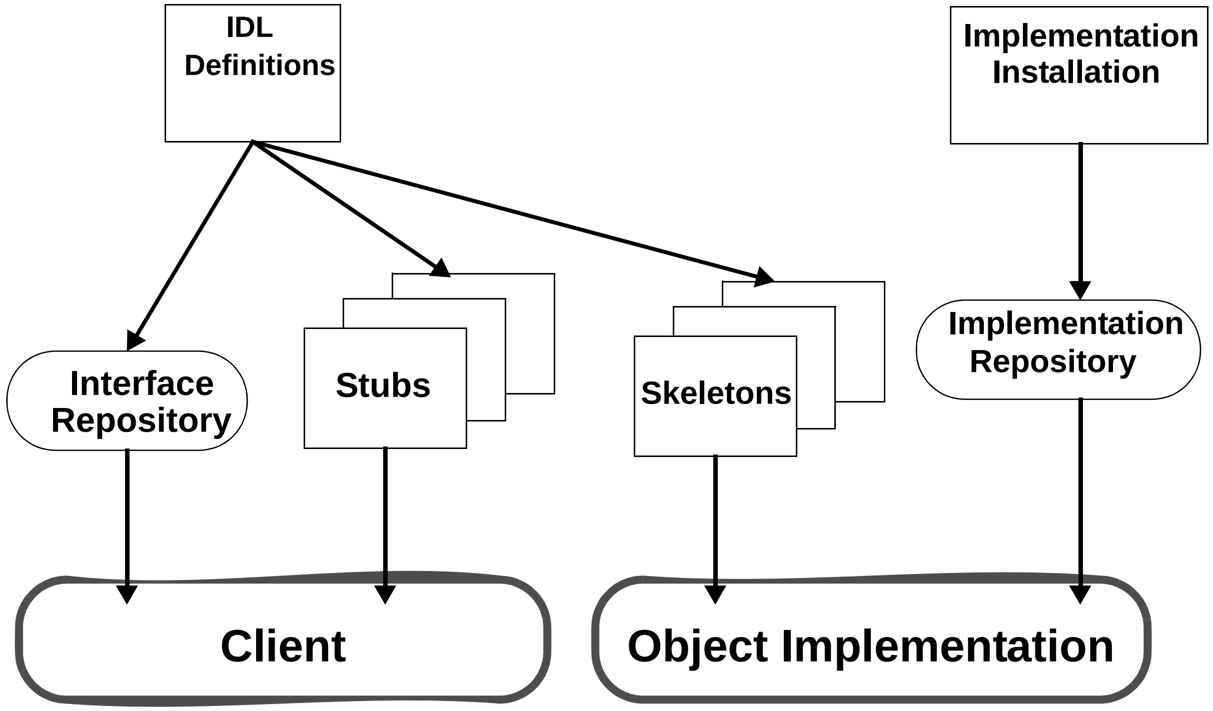

Figure 4.5- Interface and Implementation Repositories

The object implementation information is provided at installation time and is stored in the Implementation Repository for use during request delivery.

4.2.1 Object Request Broker

In the architecture, the ORB is not required to be implemented as a single component, but rather it is defined by its interfaces. Any ORB implementation that provides the appropriate interface is acceptable. The interface is organized into three categories:

1. Operations that are the same for all ORB implementations.
2. Operations that are specific to particular types of objects.
3. Operations that are specific to particular styles of object implementations.

Different ORBs may make quite different implementation choices, and, together with the IDL compilers, repositories, and various Object Adapters, provide a set of services to clients and implementations of objects that have different properties and qualities.

There may be multiple ORB implementations (also described as multiple ORBs), which have different representations for object references and different means of performing invocations. It may be possible for a client to simultaneously have access to two object references managed by different ORB implementations. When two ORBs are intended to work together, those ORBs must be able to distinguish their object references. It is not the responsibility of the client to do so.

The ORB Core is that part of the ORB that provides the basic representation of objects and communication of requests. CORBA is designed to support different object mechanisms, and it does so by structuring the ORB with components above the ORB Core, which provide interfaces that can mask the differences between ORB Cores.

4.2.2 Clients

A client of an object has access to an object reference for the object, and invokes operations on the object. A client knows only the logical structure of the object according to its interface and experiences the behavior of the object through invocations. Although we will generally consider a client to be a program or process initiating requests on an object, it is important to recognize that something is a client relative to a particular object. For example, the implementation of one object may be a client of other objects.

Clients generally see objects and ORB interfaces through the perspective of a language mapping, bringing the ORB right up to the programmer's level. Clients are maximally portable and should be able to work without source changes on any ORB that supports the desired language mapping with any object instance that implements the desired interface. Clients have no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it.

4.2.3 Object Implementations

An object implementation provides the semantics of the object, usually by defining data for the object instance and code for the object's methods. Often the implementation will use other objects or additional software to implement the behavior of the object. In some cases, the primary function of the object is to have side-effects on other things that are not objects.

A variety of object implementations can be supported, including separate servers, libraries, a program per method, an encapsulated application, an object-oriented database, etc. Through the use of additional object adapters, it is possible to support virtually any style of object implementation.

Generally, object implementations do not depend on the ORB or how the client invokes the object. Object implementations may select interfaces to ORB-dependent services by the choice of Object Adapter.

4.2.4 Object References

An Object Reference is the information needed to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping, and thus are insulated from the actual representation of them. Two ORB implementations may differ in their choice of Object Reference representations.

The representation of an object reference handed to a client is only valid for the lifetime of that client.

All ORBs must provide the same language mapping to an object reference (usually referred to as an Object) for a particular programming language. This permits a program written in a particular language to access object references independent of the particular ORB. The language mapping may also provide additional ways to access object references in a typed way for the convenience of the programmer.

There is a distinguished object reference, guaranteed to be different from all object references, that denotes no object.

4.2.5 OMG Interface Definition Language

The OMG Interface Definition Language (OMG IDL) defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. Note that although IDL provides the conceptual framework for describing the objects manipulated by the ORB, it is not necessary for there to be IDL source code available for the ORB to work. As long as the equivalent information is available in the form of stub routines or a run-time interface repository, a particular ORB may be able to function correctly.

IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

4.2.6 Mapping of OMG IDL to Programming Languages

Different object-oriented or non-object-oriented programming languages may prefer to access CORBA objects in different ways. For object-oriented languages, it may be desirable to see CORBA objects as programming language objects. Even for non-object-oriented languages, it is a good idea to hide the exact ORB representation of the object reference, method names, etc. A particular mapping of OMG IDL to a programming language should be the same for all ORB implementations. Language mapping includes definition of the language-specific data types and procedure interfaces to access objects through the ORB. It includes the structure of the client stub interface (not required for object-oriented languages), the dynamic invocation interface, the implementation skeleton, the object adapters, and the direct ORB interface.

A language mapping also defines the interaction between object invocations and the threads of control in the client or implementation. The most common mappings provide synchronous calls, in that the routine returns when the object operation completes. Additional mappings may be provided to allow a call to be initiated and control returned to the program. In such cases, additional language-specific routines must be provided to synchronize the program’s threads of control with the object invocation.

4.2.7 Client Stubs

Generally, the client stubs will present access to the OMG IDL-defined operations on an object in a way that is easy for programmers to predict once they are familiar with OMG IDL and the language mapping for the particular programming language. The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. In this case, it is necessary for the ORB and language mapping to cooperate to associate the correct stubs with the particular object reference.

4.2.8 Dynamic Invocation Interface

An interface is also available that allows the dynamic construction of object invocations, that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call or sequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed (perhaps obtaining it from an Interface Repository or other run-time source). The nature of the dynamic invocation interface may vary substantially from one programming language mapping to another.

4.2.9 Implementation Skeleton

For a particular language mapping, and possibly depending on the object adapter, there will be an interface to the methods that implement each type of object. The interface will generally be an up-call interface, in that the object implementation writes routines that conform to the interface and the ORB calls them through the skeleton.

The existence of a skeleton does not imply the existence of a corresponding client stub (clients can also make requests via the dynamic invocation interface).

It is possible to write an object adapter that does not use skeletons to invoke implementation methods. For example, it may be possible to create implementations dynamically for languages such as Smalltalk.

4.2.10 Dynamic Skeleton Interface

An interface is available, which allows dynamic handling of object invocations. That is, rather than being accessed through a skeleton that is specific to a particular operation, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's Dynamic Invocation Interface. Purely static knowledge of those parameters may be used, or dynamic knowledge (perhaps determined through an Interface Repository) may be also used, to determine the parameters.

The implementation code must provide descriptions of all the operation parameters to the ORB, and the ORB provides the values of any input parameters for use in performing the operation. The implementation code provides the values of any output parameters, or an exception, to the ORB after performing the operation. The nature of the dynamic skeleton interface may vary substantially from one programming language mapping or object adapter to another, but will typically be an up-call interface.

Dynamic skeletons may be invoked both through client stubs and through the dynamic invocation interface; either style of client request construction interface provides identical results.

4.2.11 Object Adapters

An object adapter is the primary way that an object implementation accesses services provided by the ORB. There are expected to be a few object adapters that will be widely available, with interfaces that are appropriate for specific kinds of objects. Services provided by the ORB through an Object Adapter often include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations.

The wide range of object granularities, lifetimes, policies, implementation styles, and other properties make it difficult for the ORB Core to provide a single interface that is convenient and efficient for all objects. Thus, through Object Adapters, it is possible for the ORB to target particular groups of object implementations that have similar requirements with interfaces tailored to them.

4.2.12 ORB Interface

The ORB Interface is the interface that goes directly to the ORB, which is the same for all ORBs and does not depend on the object's interface or object adapter. Because most of the functionality of the ORB is provided through the object adapter, stubs, skeleton, or dynamic invocation, there are only a few operations that are common across all objects. These operations are useful to both clients and implementations of objects.

4.2.13 Interface Repository

The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at run-time. The Interface Repository information may be used by the ORB to perform requests. Moreover, using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make an invocation on it.

In addition to its role in the functioning of the ORB, the Interface Repository is a common place to store additional information associated with interfaces to ORB objects. For example, debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects might be associated with the Interface Repository.

4.2.14 Implementation Repository

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

In addition to its role in the functioning of the ORB, the Implementation Repository is a common place to store additional information associated with implementations of ORB objects. For example, debugging information, administrative control, resource allocation, security, etc., might be associated with the Implementation Repository.

4.3 Structure of a Client

A client of an object has an object reference that refers to that object. An object reference is a token that may be invoked or passed as a parameter to an invocation on a different object. Invocation of an object involves specifying the object to be invoked, the operation to be performed, and parameters to be given to the operation or returned from it.

The ORB manages the control transfer and data transfer to the object implementation and back to the client. In the event that the ORB cannot complete the invocation, an exception response is provided. Ordinarily, a client calls a routine in its program that performs the invocation and returns when the operation is complete.

Clients access object-type-specific stubs as library routines in their program (see Figure 4.6). The client program thus sees routines callable in the normal way in its programming language. All implementations will provide a language-specific data type to use to refer to objects, often an opaque pointer. The client then passes that object reference to the stub routines to initiate an invocation. The stubs have access to the object reference representation and interact with the ORB to perform the invocation. (See the *C Language Mapping* specification for additional, general information on language mapping of object references.)

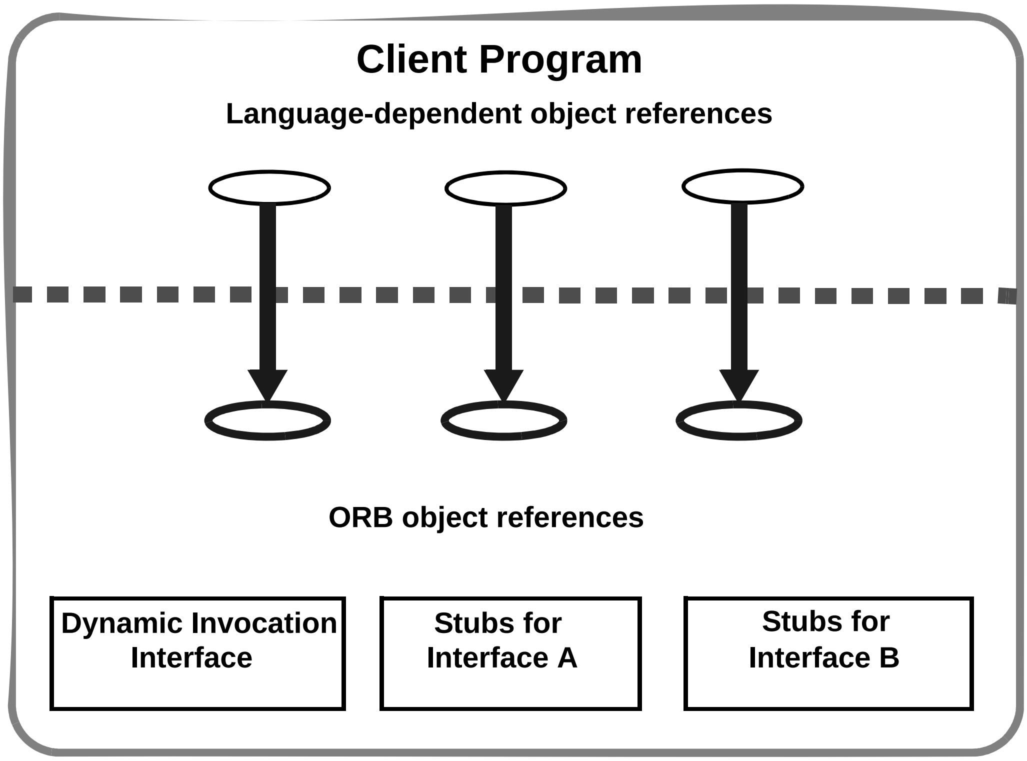

Figure 4.6- The Structure of a Typical Client

An alternative set of library code is available to perform invocations on objects, for example when the object was not defined at compile time. In that case, the client program provides additional information to name the type of the object and the method being invoked, and performs a sequence of calls to specify the parameters and initiate the invocation.

Clients most commonly obtain object references by receiving them as output parameters from invocations on other objects for which they have references. When a client is also an implementation, it receives object references as input parameters on invocations to objects it implements. An object reference can also be converted to a string that can be stored in files or preserved or communicated by different means and subsequently turned back into an object reference by the ORB that produced the string.

4.4 Structure of an Object Implementation

An object implementation provides the actual state and behavior of an object. The object implementation can be structured in a variety of ways. Besides defining the methods for the operations themselves, an implementation will usually define procedures for activating and deactivating objects and will use other objects or non-object facilities to make the object state persistent, to control access to the object, as well as to implement the methods.

The object implementation (see Figure 4.7) interacts with the ORB in a variety of ways to establish its identity, to create new objects, and to obtain ORB-dependent services. It primarily does this via access to an Object Adapter, which provides an interface to ORB services that is convenient for a particular style of object implementation.

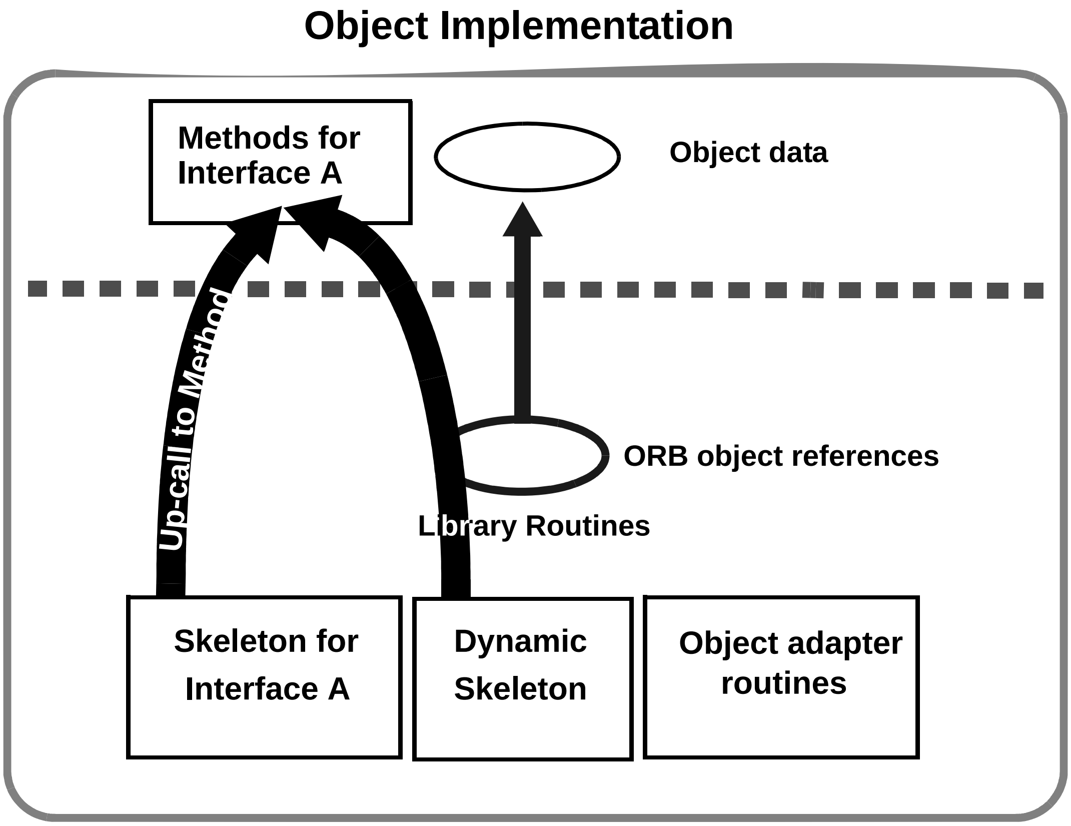

Figure 4.7- The Structure of a Typical Object Implementation

Because of the range of possible object implementations, it is difficult to be definitive about how an object implementation is structured. See the chapter on the Portable Object Adapter.

When an invocation occurs, the ORB Core, object adapter, and skeleton arrange that a call is made to the appropriate method of the implementation. A parameter to that method specifies the object being invoked, which the method can use to locate the data for the object. Additional parameters are supplied according to the skeleton definition. When the method is complete, it returns, causing output parameters or exception results to be transmitted back to the client.

When a new object is created, the ORB may be notified so that it knows where to find the implementation for that object. Usually, the implementation also registers itself as implementing objects of a particular interface, and specifies how to start up the implementation if it is not already running.

Most object implementations provide their behavior using facilities in addition to the ORB and object adapter. For example, although the Portable Object Adapter provides some persistent data associated with an object (its OID or Object ID), that relatively small amount of data is typically used as an identifier for the actual object data stored in a storage service of the object implementation’s choosing. With this structure, it is not only possible for different object implementations to use the same storage service, it is also possible for objects to choose the service that is most appropriate for them.

4.5 Structure of an Object Adapter

An object adapter (see Figure 4.8) is the primary means for an object implementation to access ORB services such as object reference generation. An object adapter exports a public interface to the object implementation, and a private interface to the skeleton. It is built on a private ORB-dependent interface.

Object adapters are responsible for the following functions:

- Generation and interpretation of object references
- Method invocation
- Security of interactions
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations
- Registration of implementations

These functions are performed using the ORB Core and any additional components necessary. Often, an object adapter will maintain its own state to accomplish its tasks. It may be possible for a particular object adapter to delegate one or more of its responsibilities to the Core upon which it is constructed.

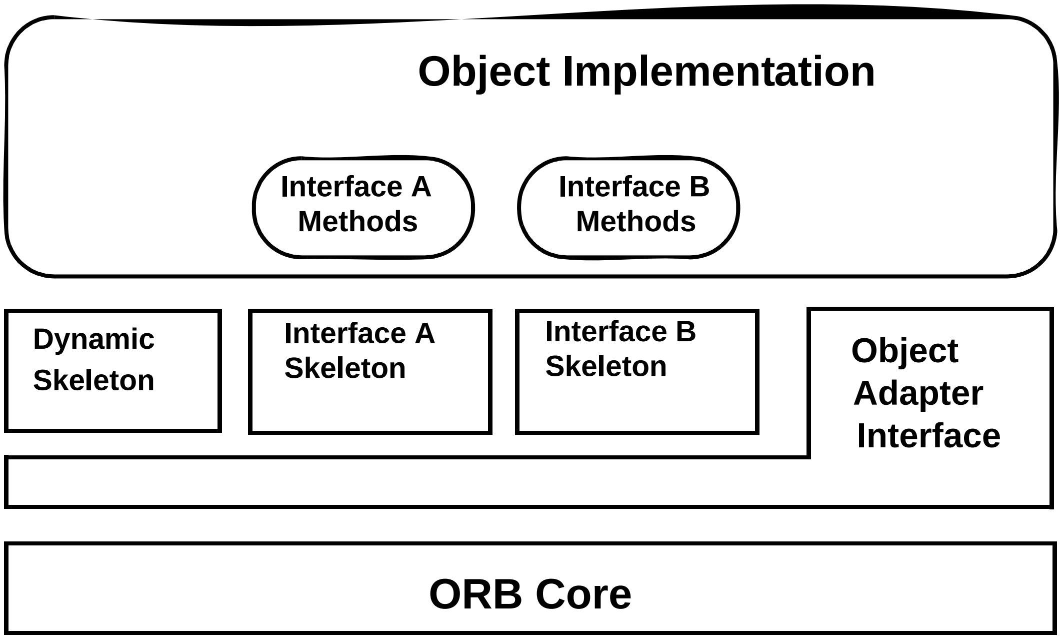

Figure 4.8- The Structure of a Typical Object Adapter

As shown in Figure 4.8, the Object Adapter is implicitly involved in invocation of the methods, although the direct interface is through the skeletons. For example, the Object Adapter may be involved in activating the implementation or authenticating the request.

The Object Adapter defines most of the services from the ORB that the Object Implementation can depend on. Different ORBs will provide different levels of service and different operating environments may provide some properties implicitly and require others to be added by the Object Adapter. For example, it is common for Object Implementations to want to store certain values in the object reference for easy identification of the object on an invocation. If the Object Adapter allows the implementation to specify such values when a new object is created, it may be able to store them in the object reference for those ORBs that permit it. If the ORB Core does not provide this feature, the Object Adapter would record the value in its own storage and provide it to the implementation on an invocation. With Object Adapters, it is possible for an Object Implementation to have access to a service whether or not it is implemented in the ORB Core — if the ORB Core provides it, the adapter simply provides an interface to it; if not, the adapter must implement it on top of the ORB Core. Every instance of a particular adapter must provide the same interface and service for all the ORBs it is implemented on.

It is also not necessary for all Object Adapters to provide the same interface or functionality. Some Object Implementations have special requirements. For example, an object-oriented database system may wish to implicitly register its many thousands of objects without doing individual calls to the Object Adapter. In such a case, it would be impractical and unnecessary for the object adapter to maintain any per-object state. By using an object adapter interface that is tuned towards such object implementations, it is possible to take advantage of particular ORB Core details to provide the most effective access to the ORB.

4.6 CORBA Required Object Adapter

There are a variety of possible object adapters; however, since the object adapter interface is something that object implementations depend on, it is desirable that there be as few as practical. Most object adapters are designed to cover a range of object implementations, so only when an implementation requires radically different services or interfaces should a new object adapter be considered. In this section, we briefly describe the object adapter defined in this specification.

4.6.1 Portable Object Adapter

This specification defines a Portable Object Adapter that can be used for most ORB objects with conventional implementations. (See the *Portable Object Adapter* chapter for more information.) The intent of the POA, as its name suggests, is to provide an Object Adapter that can be used with multiple ORBs with a minimum of rewriting needed to deal with different vendors' implementations.

This specification allows several ways of using servers but it does not deal with the administrative issues of starting server programs. Once started, however, there can be a servant started and ended for a single method call, a separate servant for each object, or a shared servant for all instances of the object type. It allows for groups of objects to be associated by means of being registered with different instances of the POA object and allows implementations to specify their own activation techniques. If the implementation is not active when an invocation is performed, the POA will start one. The POA is specified in IDL, so its mapping to languages is largely automatic, following the language mapping rules. (The primary task left for a language mapping is the definition of the Servant type.)

4.7 The Integration of Foreign Object Systems

The Common ORB Architecture is designed to allow interoperation with a wide range of object systems (see Figure 4.9). Because there are many existing object systems, a common desire will be to allow the objects in those systems to be accessible via the ORB. For those object systems that are ORBs themselves, they may be connected to other ORBs through the mechanisms described throughout this manual.

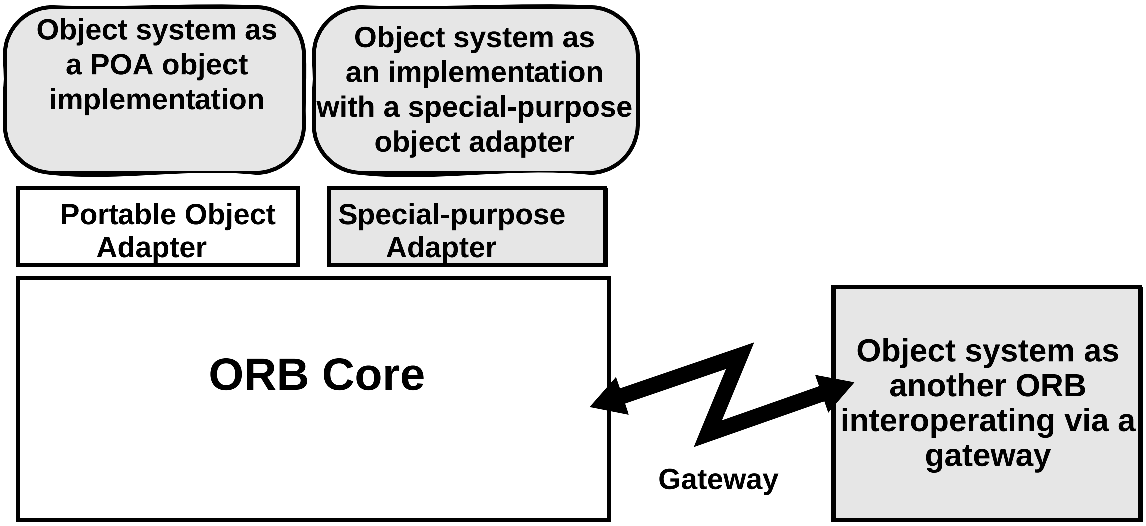
Object system as a POA object implementation
Object system as an implementation with a special-purpose object adapter
Portable Object Adapter
Special-purpose Adapter
ORB Core
Gateway
Object system as another ORB interoperating via a gateway

Figure 4.9- Different Ways to Integrate Foreign Object Systems

For object systems that simply want to map their objects into ORB objects and receive invocations through the ORB, one approach is to have those object systems appear to be implementations of the corresponding ORB objects. The object system would register its objects with the ORB and handle incoming requests, and could act like a client and perform outgoing requests.

In some cases, it will be impractical for another object system to act like a POA object implementation. An object adapter could be designed for objects that are created in conjunction with the ORB and that are primarily invoked through the ORB. Another object system may wish to create objects without consulting the ORB, and might expect most invocations to occur within itself rather than through the ORB. In such a case, a more appropriate object adapter might allow objects to be implicitly registered when they are passed through the ORB.

4.8 Interoperability Model

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs. The approach to “interORBability” is universal, because its elements can be combined in many ways to satisfy a very broad range of needs.

4.8.1 Elements of Interoperability

The elements of interoperability are as follows:

- ORB interoperability architecture
- Inter-ORB bridge support
- General and Internet inter-ORB Protocols (GIOPs and IIOPs)

In addition, the architecture accommodates environment-specific inter-ORB protocols (ESIOPs) that are optimized for particular environments such as DCE.

4.8.2 ORB Interoperability Architecture

The ORB Interoperability Architecture provides a conceptual framework for defining the elements of interoperability and for identifying its compliance points. It also characterizes new mechanisms and specifies conventions necessary to achieve interoperability between independently produced ORBs.

Specifically, the architecture introduces the concepts of *immediate* and *mediated bridging* of ORB domains. The Internet Inter-ORB Protocol (IIOP) forms the common basis for broad-scope mediated bridging. The inter-ORB bridge support can be used to implement both immediate bridges and to build “half-bridges” to mediated bridge domains.

By use of bridging techniques, ORBs can interoperate without knowing any details of that ORB’s implementation, such as what particular IPC or protocols (such as ESIOPs) are used to implement the *CORBA* specification.

The IIOP may be used in bridging two or more ORBs by implementing “half bridges” that communicate using the IIOP. This approach works for both stand-alone ORBs, and networked ones that use an ESIOP.

The IIOP may also be used to implement an ORB’s internal messaging, if desired. Since ORBs are not required to use the IIOP internally, the goal of not requiring prior knowledge of each others’ implementation is fully satisfied.

4.8.3 Inter-ORB Bridge Support

The interoperability architecture clearly identifies the role of different kinds of domains for ORB-specific information. Such domains can include object reference domains, type domains, security domains (e.g., the scope of a *Principal* identifier), a transaction domain, and more.

Where two ORBs are in the same domain, they can communicate directly. In many cases, this is the preferable approach. This is not always true, however, since organizations often need to establish local control domains.

When information in an invocation must leave its domain, the invocation must traverse a bridge. The role of a bridge is to ensure that content and semantics are mapped from the form appropriate to one ORB to that of another, so that users of any given ORB only see their appropriate content and semantics.

The inter-ORB bridge support element specifies ORB APIs and conventions to enable the easy construction of interoperability bridges between ORB domains. Such bridge products could be developed by ORB vendors, Sieves, system integrators, or other third-parties.

Because the extensions required to support Inter-ORB Bridges are largely general in nature, do not impact other ORB operation, and can be used for many other purposes besides building bridges, they are appropriate for all ORBs to support. Other applications include debugging, interposing of objects, implementing objects with interpreters and scripting languages, and dynamically generating implementations.

The inter-ORB bridge support can also be used to provide interoperability with non-CORBA systems, such as Microsoft’s Component Object Model (COM). The ease of doing this will depend on the extent to which those systems conform to the CORBA Object Model.

4.8.4 General Inter-ORB Protocol (GIOP)

The General Inter-ORB Protocol (GIOP) element specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs. The GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions.

It does not require or rely on the use of higher level RPC mechanisms. The protocol is simple, scalable, and relatively easy to implement. It is designed to allow portable implementations with small memory footprints and reasonable performance, with minimal dependencies on supporting software other than the underlying transport layer.

While versions of the GIOP running on different transports would not be directly interoperable, their commonality would allow easy and efficient bridging between such networking domains.

4.8.5 Internet Inter-ORB Protocol (IIOP)®

The Internet Inter-ORB Protocol (IIOP)® element specifies how GIOP messages are exchanged using TCP/IP connections. The IIOP specifies a standardized interoperability protocol for the Internet, providing “out of the box” interoperation with other compatible ORBs based on the most popular product- and vendor-neutral transport layer. It can also be used as the protocol between half-bridges (see below).

The protocol is designed to be suitable and appropriate for use by any ORB to interoperate in Internet Protocol domains unless an alternative protocol is necessitated by the specific design center or intended operating environment of the ORB. In that sense it represents the basic inter-ORB protocol for TCP/IP environments, a most pervasive transport layer.

The IIOP’s relationship to the GIOP is similar to that of a specific language mapping to OMG IDL; the GIOP may be mapped onto a number of different transports, and specifies the protocol elements that are common to all such mappings. The GIOP by itself, however, does not provide complete interoperability, just as IDL cannot be used to build complete programs. The IIOP and other similar mappings to different transports, are concrete realizations of the abstract GIOP definitions, as shown in Figure 4.10.

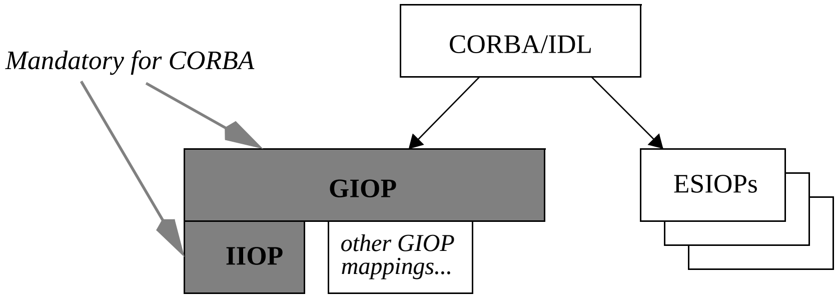

Figure 4.10- Inter-ORB Protocol Relationships

4.8.6 Environment-Specific Inter-ORB Protocols (ESIOPs)

This specification also makes provision for an open-ended set of Environment-Specific Inter-ORB Protocols (ESIOPs). Such protocols would be used for “out of the box” interoperation at user sites where a particular networking or distributing computing infrastructure is already in general use.

Because of the opportunity to leverage and build on facilities provided by the specific environment, ESIOPs might support specialized capabilities such as those relating to security and administration.

While ESIOPs may be optimized for particular environments, all ESIOP specifications will be expected to conform to the general ORB interoperability architecture conventions to enable easy bridging. The inter-ORB bridge support enables bridges to be built between ORB domains that use the IIOP and ORB domains that use a particular ESIOP.

4.8.7 Relationship to Previous Versions of CORBA

The ORB Interoperability Architecture builds on Common Object Request Broker Architecture by adding the notion of ORB Services and their domains. (ORB Services are described in 13.2, ’Goals of the General Inter-ORB Protocol’). The architecture defines the problem of ORB interoperability in terms of bridging between those domains, and defines several ways in which those bridges can be constructed. The bridges can be internal (in-line) and external (request-level) to ORBs.

APIs included in the interoperability specifications include compatible extensions to previous versions of *CORBA* to support request-level bridging:

- A Dynamic Skeleton Interface (DSI) is the basic support needed for building request-level bridges. It is the server-side analogue of the Dynamic Invocation Interface and in the same way it has general applicability beyond bridging. For information about the Dynamic Skeleton Interface, refer to the *Dynamic Skeleton Interface* chapter.
- APIs for managing object references have been defined, building on the support identified for the Relationship Service. The APIs are defined in Object Reference Operations in the *ORB Interface* chapter of this book. The Relationship Service is described in the *Relationship Service* specification; refer to the *CosObjectIdentity Module* section of that specification.

4.8.8 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 domain, and other “run-time” characteristics of a system. Technology domains identify common protocols, syntaxes, and similar “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.

4.9 Quality of Service

4.9.1 QoS Abstract Model Design

This section describes each of the components in the Quality of Service (QoS) abstract model and their relationships. The specification defines a framework within which current QoS levels are queried and overridden. This framework is intended to be of use for CORBAServices specifiers, as well as for future revisions of CORBA. The Messaging-specific QoS are defined in terms of this framework.

Note – The QoS definitions specified in this specification are applied to both synchronous as well as asynchronous invocations.

4.9.2 Model Components

The QoS framework abstract model consists of the following components:

- **Policy** - The base interface from which all QoS objects derive.
- **PolicyList** - A sequence of Policy objects.
- **PolicyManager** - An interface with operations for querying and overriding QoS **Policy** settings.
    - Mechanisms for obtaining **Policy** override management operations at each relevant application scope:
    - The ORB's **PolicyManager** is obtained through invoking **ORB::resolve_initial_references** with the **ObjectId** "ORBPolicyManager."
    - A **CORBA::PolicyCurrent** derived from **CORBA::Current** is used for managing the thread's QoS Policies. A reference to this interface is obtained through an invocation of **ORB::resolve_initial_references** with the **ObjectId** "PolicyCurren."
    - Accessor operations on **CORBA::Object** allow querying and overriding of QoS at the object reference scope.
    - The application of QoS on a Portable Object Adapter is done through the currently existing mechanism of passing a **PolicyList** to the **POA::create_POA** operation.
- Mechanisms for transporting Policy values as part of interoperable object references and within requests:
    - **TAG_POLICIES** - A Profile Component containing the sequence of QoS policies exported with the object reference by an object adapter.
    - **INVOCATION_POLICIES** - A Service Context containing a sequence of QoS policies in effect for the invocation.

The Messaging QoS abstract model consists of a number of **CORBA::Policy**-derived interfaces:

- Client-side Policies are applied to control the behavior of requests and replies. These include Priority, RequestEndTime, and Queueing QoS.
- Server-side Policies are applied to control the default behavior of invocations on a target. These include QueueOrder and Transactionality QoS.

4.9.3 Component Relationships

Programmers set QoS at various levels of scope by creating a Policy-derived Messaging QoS Policy and selecting the interface for the particular scope. It is anticipated that the following is the standard use-case scenario:

- A POA is created with a certain set of QoS. When object references are created by that POA, the required and supported QoS are encoded in that object reference. Such an object reference is then exported for use by a client.
- Within a client, the ORB's **PolicyManager** interface is obtained to set QoS for the entire ORB (for the entire process when only one ORB is present) either programmatically, or administratively. The Policies set here are valid for all invocations in the process. A programmer-constructed **PolicyList** is used with this interface to actually set the QoS.

- Within that same client, the **CORBA::PolicyCurrent** is obtained to set QoS for all invocations in the current thread. This interface is derived from the **PolicyManager** interface, which can be used to change the QoS for each invocation. A programmer-constructed **PolicyList** is used with this interface to actually set the QoS.
- Within that same client, the object reference is obtained and an invocation of its **get_client_policy** operation queries the most specific QoS overrides. A programmer-constructed **PolicyList** may be passed to the Object's **set_policy_overrides** operation to obtain a new Object reference with revised QoS. Setting the QoS here applies to all invocations using the new Object reference and supersedes (if possible) those set at the ORB and thread (Current) scopes. The current set of overrides can be validated by calling the Object's pseudo-operation **validate_connection**, which will attempt to locate a target for the object reference if no target has yet been located. At this time, any Policy overrides placed at the Object, Thread or ORB scope will be reconciled with the QoS Policies established for that object reference when it was created by the POA. The current *effective* **Policy** can then be queried by invoking **get_policy**, which returns the **Policy** value that is in effect.
- Unseen by the application, the ORB (including the protocol engine) modifies its internal behavior in order to realize the quality of service indicated by the client through the first three steps.

4.9.4 Component Design

Design decisions were made with respect to the following components of the QoS framework:

- Each QoS is an interface derived from **CORBA::Policy**. The design trade-offs focused on ease of application interface for setting specific QoS values, extensibility for new QoS types and values, and compactness so the QoS values can be represented efficiently in Service Contexts and IOR Profile Components. Several alternatives were considered as the basic type for each QoS entity before the decision was made to use the **Policy** interface:
  - **CORBA::NamedValue** - A pair of **string** and **any** were considered mainly due to the flexibility afforded by using an any to represent QoS values. This design was discounted due to the untyped nature of the **any** and the application development and execution costs of inserting typed data into and extracting typed data from values of type **any**. Furthermore, the presence of a full typecode within an any makes the size of such pairs too large for inclusion in compact Service Contexts and Profile Components.
  - Stateful CORBA **valuetype** - Although the **valuetype** does present a typed interface to the application program, including **valuetype**s in Service Contexts and IOR Profile Components is too expensive due to the presence of full repository identifier information when the **valuetype** is marshaled. Furthermore, there are issues associated with potential truncation of such QoS **valuetype**s when passed as formal arguments of their base type.
  - Interfaces derived from **CORBA::Policy** and compact representation. In the model chosen by this specification, the QoS values are accessible through locality-constrained interfaces. Derivation from **CORBA::Policy** allows reuse of existing interfaces and operations for policy management. When certain QoS values must be marshaled in a Service Context or an IOR Profile Component, the most compact format was chosen. The type of QoS **Policy** represented is indicated by a structure containing the integral **PolicyType** and a **sequence** of **octet** holding the values for that policy.
- A generic factory for creating QoS Policies. In the *POA* specification within *CORBA*, each POA **Policy** is created through an operation on the POA itself. Although this presents a convenient typed interface for the creation of **Policy** objects, it causes serious problems when new POA Policies are introduced. To fit with the current model, operations would have to be added to the POA interface for every new type of POA **Policy**. To address this potential administrative nightmare, this specification introduces a new ORB operation **create_policy**. Rather than introducing typed operations for creating all of the Messaging QoS Policies discussed in this specification, the generic factory operation is used.
- A **RebindPolicy** client-side QoS **Policy** to ensure deterministic effective QoS. In *CORBA*, *transparent rebinding* of an object reference may take place during any invocation. Rebinding is defined here to mean changing the client-

visible QoS as a result of replacing the IOR Profile used by a client's object reference with a new IOR Profile. Transparent rebinding is defined as when this happens without notice to the client application. Typically, this happens within GIOP through the use of location forwarding. The default **RebindPolicy** (and the only *CORBA* behavior) supports this transparent rebind. For an application with rigorous quality of service requirements, such transparent rebinding can cause problems. For instance, unexpected errors may occur if the application sets its QoS Policies appropriately for an object reference, and then the ORB transparently changes the application's assumptions about that reference by obtaining a new IOR. The **RebindPolicy** has been added so that applications can prevent the ORB from silently changing the IOR Profile (and therefore the server-side QoS) that have been assumed. A more rigorous value of this **Policy** even precludes the ORB from silently closing and opening connections (when IIOP is being used, for example). The specific requirements demanded by an application dictate which level of **RebindPolicy** is necessary.

5 The Object Model

This chapter describes the concrete object model that underlies the CORBA architecture. The model is derived from the abstract Core Object Model defined by the Object Management Group in the *Object Management Architecture Guide*. (Information about the *OMA Guide* and other books in the CORBA documentation set is provided in this document's preface.)

Compliance

This chapter contains non-normative descriptive material.

5.1 Overview

The object model provides an organized presentation of object concepts and terminology. It defines a partial model for computation that embodies the key characteristics of objects as realized by the conformant technologies. The OMG object model is *abstract* in that it is not directly realized by any particular technology. The model described here is a *concrete* object model. A concrete object model may differ from the abstract object model in several ways:

- It may *elaborate* the abstract object model by making it more specific; for example, by defining the form of request parameters or the language used to specify types.
- It may *populate* the model by introducing specific instances of entities defined by the model; for example, specific objects, specific operations, or specific types.
- It may *restrict* the model by eliminating entities or placing additional restrictions on their use.

An object system is a collection of objects that isolates the requestors of services (clients) from the providers of services by a well-defined encapsulating interface. In particular, clients are isolated from the implementations of services as data representations and executable code.

The object model first describes concepts that are meaningful to clients, including such concepts as object creation and identity, requests and operations, types and signatures. It then describes concepts related to object implementations, including such concepts as methods, execution engines, and activation.

The object model is most specific and prescriptive in defining concepts meaningful to clients. The discussion of object implementation is more suggestive, with the intent of allowing maximal freedom for different object technologies to provide different ways of implementing objects.

There are some other characteristics of object systems that are outside the scope of the object model. Some of these concepts are aspects of application architecture, some are associated with specific domains to which object technology is applied. Such concepts are more properly dealt with in an architectural reference model. Examples of excluded concepts are compound objects, links, copying of objects, change management, and transactions. Also outside the scope of the object model are the details of control structure: the object model does not say whether clients and/or servers are single-threaded or multi-threaded, and does not specify how event loops are programmed nor how threads are created, destroyed, or synchronized.

This object model is an example of a *classical object model*, where a client sends a message to an object. Conceptually, the object interprets the message to decide what service to perform. In the classical model, a message identifies an object and zero or more actual parameters. As in most classical object models, a distinguished first parameter is required, which identifies the operation to be performed; the interpretation of the message by the object involves selecting a method based on the specified operation. Operationally, of course, method selection could be performed either by the object or the ORB.

5.2 Foundations of the Model

An Object Request Broker is built from three foundational concepts.

5.2.1 Object

An object system includes entities known as objects. An *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

5.2.2 Request

Object interact when a clients request a service by issuing requests.

The term *request* is broadly used to refer to the entire sequence of causally related events that transpires between a client initiating it and the last event causally associated with that initiation. For example:

- the client receives the final response associated with that *request* from the server,
- the server carries out the associated operation in case of a oneway request, or
- the sequence of events associated with the *request* terminates in a failure of some sort. The initiation of a Request is an event.

The information associated with a request consists of an operation, a target object, zero or more (actual) parameters, and an optional request context.

5.2.3 Broker

The broker is the infrastructure in place that enables a client to interact with, i.e., request a service of, an object. It is the intermediary between client and (server) object.

5.3 Constituents of the Model

5.3.1 Client

An object system provides services to clients. A *client* of a service is any entity capable of requesting the service. Object semantics are the concepts relevant to clients.

5.3.2 Object Reference

An *object reference* is a value that reliably denotes a particular object. Specifically, an object reference will identify the same object each time the reference is used in a request (subject to certain pragmatic limits of space and time). An object may be denoted by multiple, distinct object references.

5.3.3 Request Terminations

A request causes a service to be performed on behalf of the client. There are a number of possible outcomes of a request, called *terminations*. One possible termination of performing a service is returning results to the client. If there are no results defined for the request, then the termination represents synchronization between server and client.

If an abnormal condition occurs during the performance of a request, an exceptional termination occurs. The termination may carry additional return parameters particular to that exception.

5.3.4 Request Parameters

A request may have parameters that are used to pass data to the target object; it may also have a request context that provides additional information about the request. A request context is a mapping from strings to strings.

The request parameters are identified by position. A parameter may be an input parameter, an output parameter, or an input-output parameter. A request may also return a single *return result value*, as well as the results stored into the output and input-output parameters.

The following semantics hold for all requests:

- Any aliasing of parameter values is neither guaranteed removed nor guaranteed to be preserved.
- The order in which aliased output parameters are written is not guaranteed.
- The return result and the values stored into the output and input-output parameters are undefined if an exception is returned.

A *value* is anything that may be a legitimate (actual) parameter in a request. More particularly, a value is an instance of an OMG IDL data type. There are non-object values, as well as values that reference objects.

5.3.5 Object Lifecycle

Objects can be created and destroyed. From a client’s point of view, there is no special mechanism for creating or destroying an object. Objects are created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object.

5.3.6 Types

A *type* is an identifiable entity with an associated predicate (a single-argument mathematical function with a boolean result) defined over entities. An entity *satisfies* a type if the predicate is true for that entity. An entity that satisfies a type is called a *member of the type*.

Types are used in signatures to restrict a possible parameter or to characterize a possible result.

The *extension of a type* is the set of entities that satisfy the type at any particular time.

An *object type* is a type whose members are object references. In other words, an object type is satisfied only by object references.

Constraints on the data types in this model are shown in this section.

5.3.6.1 Basic types

- 16-bit, 32-bit, and 64-bit signed and unsigned 2’s complement integers.
- Single-precision (32-bit), double-precision (64-bit), and double-extended (a mantissa of at least 64 bits, a sign bit and an exponent of at least 15 bits) IEEE floating point numbers.
- Fixed-point decimal numbers of up to 31 significant digits.
- Characters, as defined in ISO Latin-1 (8859.1) and other single- or multi-byte character sets.
- A boolean type taking the values TRUE and FALSE.
- An 8-bit opaque detectable, guaranteed to *not* undergo any conversion during transfer between systems.
- Enumerated types consisting of ordered sequences of identifiers.
- A string type, which consists of a variable-length array of characters; the length of the string is a non-negative integer, and is available at run-time. The length may have a maximum bound defined.
- A wide character string type, which consists of a variable-length array of (fixed width) wide characters; the length of the wide string is a non-negative integer, and is available at run-time. The length may have a maximum bound defined.
- A container type “any,” which can represent any possible basic or constructed type.
- Wide characters that may represent characters from any wide character set.
- Wide character strings, which consist of a length, available at runtime, and a variable-length array of (fixed width) wide characters.

5.3.6.2 Constructed types

- A record type (called struct), which consists of an ordered set of (name,value) pairs.
- A discriminated union type, which consists of a discriminator (whose exact value is always available) followed by an instance of a type appropriate to the discriminator value.
- A sequence type, which consists of a variable-length array of a single type; the length of the sequence is available at run-time.
- An array type, which consists of a fixed-shape multidimensional array of a single type.
- An interface type, which specifies the set of operations that an instance of that type must support.
- A value type, which specifies state as well as a set of operations that an instance of that type must support.

Entities in a request are restricted to values that satisfy these type constraints. The legal entities are shown in Figure 5.1. No particular representation for entities is defined.

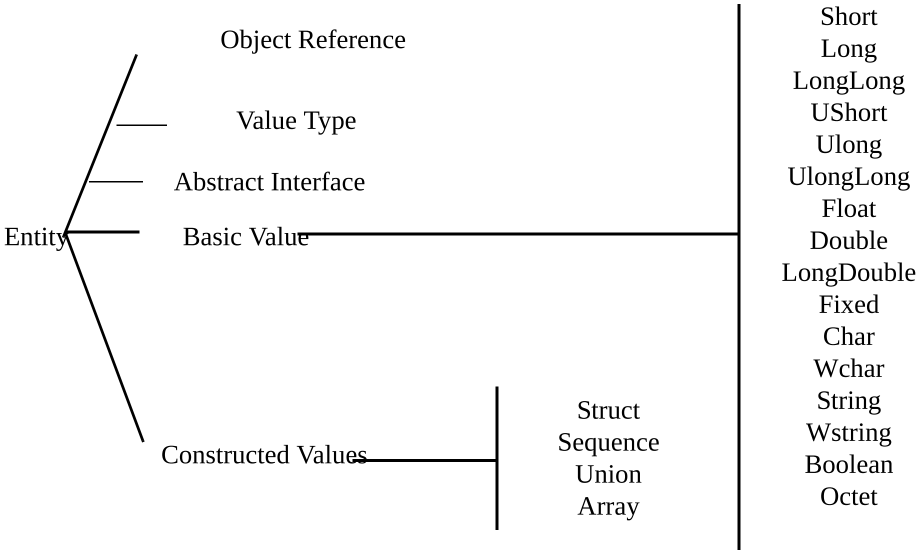

Figure 5.1- Legal Values

5.3.7 Interface

An *interface* is a description of a set of possible operations that a client may request of an object, through that interface. It provides a syntactic description of how a service provided by an object supporting this interface, is accessed via this set of operations. An object *satisfies* an interface if it provides its service through the operations of the interface according to the specification of the operations (see Section 5.3.10, “Operation,” on page 30).

The *interface type* for a given interface is an object type, such that an object reference will satisfy the type, if and only if the referent object also satisfies the interface.

Interfaces are specified in OMG IDL. Interface inheritance provides the composition mechanism for permitting an object to support multiple interfaces. The *principal interface* is simply the most-specific interface that the object supports, and consists of all operations in the transitive closure of the interface inheritance graph.

Interfaces satisfy the Liskov substitution principle. If interface A is derived from interface B, then a reference to an object that supports interface A can be used where the formal type of a parameter is declared to be B.

5.3.8 Value Type

A *value type* is an entity, which shares many of the characteristics of interfaces and structs. It is a description of both a set of operations that a client may request and of state that is accessible to a client. Instances of a value type are always local concrete implementations in some programming language.

A value type, in addition to the operations and state defined for itself, may also inherit from other value types, and through multiple inheritance support other interfaces.

Value types are specified in OMG IDL.

An *abstract value types* describes a value type that is a “pure” bundle of operations with no state.

5.3.9 Abstract Interface

An *abstract interface* is an entity, which may at runtime represent either a regular interface or a value type. Like an abstract value type, it is a pure bundle of operations with no state. Unlike an abstract value type, it does not imply pass-by-value semantics, and unlike a regular interface type, it does not imply pass-by-reference semantics. Instead, the entity's runtime type determines which of these semantics are used.

5.3.10 Operation

An *operation* is an identifiable entity that denotes the indivisible primitive of service provision that can be requested. The act of requesting an operation is referred to as *invoking the operation*. An operation is identified by an *operation identifier.*

An operation has a *signature* that describes the legitimate values of request parameters and returned results. In particular, a *signature* consists of:

- A specification of the parameters required in requests for that operation.
- A specification of the result of the operation.
- An identification of the user exceptions that may be raised by an invocation of the operation.
- A specification of additional contextual information that may affect the invocation.
- An indication of the execution semantics the client should expect from an invocation of the operation.

Operations are (potentially) *generic*, meaning that a single operation can be uniformly invoked on objects with different implementations, possibly resulting in observably different behavior. Genericity is achieved in this model via interface inheritance in IDL and the total decoupling of implementation from interface specification.

5.3.11 Operation Signature

The general form for an operation signature is:

```
[oneway] <op_type_spec> <identifier> (param1, ..., paramL)
    [raises(except1,...,exceptN)] [context(name1, ..., nameM)]
```

where:

- The optional **oneway** keyword indicates that best-effort semantics are expected of requests for this operation; the default semantics are exactly-once if the operation successfully returns results or at-most-once if an exception is returned.
- The **<op_type_spec>** is the type of the return result.
- The **<identifier>** provides a name for the operation in the interface.
- The operation parameters needed for the operation; they are flagged with the modifiers **in**, **out**, or **inout** to indicate the direction in which the information flows (with respect to the object performing the request).
- The optional **raises** expression indicates which user-defined exceptions can be signaled to terminate an invocation of this operation; if such an expression is not provided, no user-defined exceptions will be signaled.

- The optional **context** expression indicates which request context information will be available to the object implementation; no other contextual information is required to be transported with the request.

5.3.12 Parameters

A parameter is characterized by its mode and its type. The *mode* indicates whether the value should be passed from client to server (**in**), from server to client (**out**), or both (**inout**). The parameter's type constrains the possible value, which may be passed in the directions dictated by the mode.

5.3.13 Return Result

The return result is a distinguished **out** parameter.

5.3.14 Exceptions

An *exception* is an indication that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information.

The additional, exception-specific information is a specialized form of record. As a record, it may consist of any of the types described in Section 5.3.6, "Types," on page 27.

All signatures implicitly include the system exceptions; the standard system exceptions are described in Section 5.6.2, "System Exceptions," on page 24.

5.3.15 Contexts

A *request context* provides additional, operation-specific information that may affect the performance of a request.

5.3.16 Execution Semantics

Two styles of execution semantics are defined by the object model:

- At-most-once: if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once.
- Best-effort: a best-effort operation is a request-only operation (i.e., it cannot return any results and the requester never synchronizes with the completion, if any, of the request).

The execution semantics to be expected is associated with an operation. This prevents a client and object implementation from assuming different execution semantics.

Note that a client is able to invoke an at-most-once operation in a synchronous or deferred-synchronous manner.

5.3.17 Attributes

An interface may have attributes. An attribute is logically equivalent to declaring a pair of accessor functions: one to retrieve the value of the attribute and one to set the value of the attribute.

An attribute may be read-only, in which case only the retrieval accessor function is defined.

5.4 Facets of the Model

This section defines the concepts associated with object implementation (i.e., the concepts relevant to realizing the behavior of objects in a computational system).

The implementation of an object system carries out the computational activities needed to effect the behavior of requested services. These activities may include computing the results of the request and updating the system state. In the process, additional requests may be issued.

The implementation model consists of two parts: the execution model and the construction model. The execution model describes how services are performed. The construction model describes how services are defined.

5.4.1 The Execution Model: Performing Services

A requested service is performed in a computational system by executing code that operates upon some data. The data represents a component of the state of the computational system. The code performs the requested service, which may change the state of the system.

Code that is executed to perform a service is called a *method*. A method is an immutable description of a computation that can be interpreted by an execution engine. A method has an immutable attribute called a *method format* that defines the set of execution engines that can interpret the method. An *execution engine* is an abstract machine (not a program) that can interpret methods of certain formats, causing the described computations to be performed. An execution engine defines a dynamic context for the execution of a method. The execution of a method is called a *method activation*.

When a client issues a request, a method of the target object is called. The input parameters passed by the requestor are passed to the method and the output and input-output parameters and return result value (or exception and its parameters) are passed back to the requestor.

Performing a requested service causes a method to execute that may operate upon an object's persistent state. If the persistent form of the method or state is not accessible to the execution engine, it may be necessary to first copy the method or state into an execution context. This process is called *activation*; the reverse process is called *deactivation*.

5.4.2 The Construction Model

A computational object system must provide mechanisms for realizing behavior of requests. These mechanisms include definitions of object state, definitions of methods, and definitions of how the object infrastructure is to select the methods to execute and to select the relevant portions of object state to be made accessible to the methods. Mechanisms must also be provided to describe the concrete actions associated with object creation, such as association of the new object with appropriate methods.

An *object implementation*—or *implementation*, for short—is a definition that provides the information needed to create an object and to allow the object to participate in providing an appropriate set of services. An implementation typically includes, among other things, definitions of the methods that operate upon the state of an object. It also typically includes information about the intended types of the object.

5.4.3 The Interoperability Model

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs. The approach to "interORBability" is universal, because its elements can be combined in many ways to satisfy a very broad range of needs.

The ORB Interoperability Architecture provides a conceptual framework for defining the elements of interoperability and for identifying its compliance points. It also characterizes new mechanisms and specifies conventions necessary to achieve interoperability between independently produced ORBs.

5.4.3.1 General Inter-ORB Protocol (GIOP)

The General Inter-ORB Protocol (GIOP) element specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs. The GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions. It does not require or rely on the use of higher level RPC mechanisms. The protocol is simple, scalable and relatively easy to implement. It is designed to allow portable implementations with small memory footprints and reasonable performance, with minimal dependencies on supporting software other than the underlying transport layer.

While versions of the GIOP running on different transports would not be directly interoperable, their commonality would allow easy and efficient bridging between such networking domains.

5.4.3.2 Internet Inter-ORB Protocol (IIOP)®

The Internet Inter-ORB Protocol (IIOP)® element specifies how GIOP messages are exchanged using TCP/IP connections. The IIOP specifies a standardized interoperability protocol for the Internet, providing “out of the box” interoperation with other compatible ORBs based on the most popular product- and vendor-neutral transport layer. It can also be used as the protocol between half-bridges (see below).

The protocol is designed to be suitable and appropriate for use by any ORB to interoperate in Internet Protocol domains unless an alternative protocol is necessitated by the specific design center or intended operating environment of the ORB. In that sense it represents the basic inter-ORB protocol for TCP/IP environments, a most pervasive transport layer.

The IIOP’s relationship to the GIOP is similar to that of a specific language mapping to OMG IDL; the GIOP may be mapped onto a number of different transports, and specifies the protocol elements that are common to all such mappings. The GIOP by itself, however, does not provide complete interoperability, just as IDL cannot be used to build complete programs. The IIOP and other similar mappings to different transports, are concrete realizations of the abstract GIOP definitions, as shown in Figure 5.2.

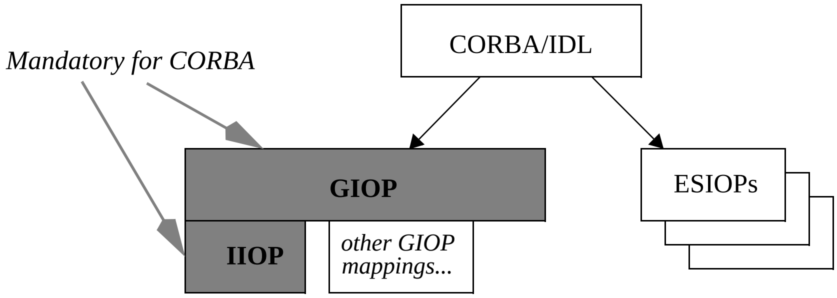

Figure 5.2- Inter-ORB Protocol Relationships

5.4.3.3 Environment-Specific Inter-ORB Protocols (ESIOPs)

This specification also makes provision for an open-ended set of Environment-Specific Inter-ORB Protocols (ESIOPs). Such protocols would be used for “out of the box” interoperation at user sites where a particular networking or distributing computing infrastructure is already in general use.

Because of the opportunity to leverage and build on facilities provided by the specific environment, ESIOPs might support specialized capabilities such as those relating to security and administration.

While ESIOPs may be optimized for particular environments, all ESIOP specifications will be expected to conform to the general ORB interoperability architecture conventions to enable easy bridging. The inter-ORB bridge support enables bridges to be built between ORB domains that use the IIOP and ORB domains that use a particular ESIOP.

5.4.3.4 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, syntaxe, and similar “build-time” characteristics. In many cases, the need for technology domains derives from basic requirements of administrative domains.

5.4.4 The Federation Model: Interworking

The Federation Model provides a framework for specifying the interworking between dislike object systems. If these systems support GIOP but are not otherwise fully conformant with the CORBA architecture they can interwork by using bridges or half-bridges. If the interaction protocol is itself different between two system, then a gateway must be constructed.

5.4.4.1 Inter-ORB Bridges

The interoperability architecture clearly identifies the role of different kinds of domains for ORB-specific information. Such domains can include object reference domains, type domains, security domains (e.g., the scope of a *Principal* identifier), a transaction domain, and more.

Where two ORBs are in the same domain, they can communicate directly. In many cases, this is the preferable approach. This is not always true, however, since organizations often need to establish local control domains.

When information in an invocation must leave its domain, the invocation must traverse a bridge. The role of a bridge is to ensure that content and semantics are mapped from the form appropriate to one ORB to that of another, so that users of any given ORB only see their appropriate content and semantics.

A bridging architecture can be used for other applications, including debugging, interposing of objects, implementing objects with interpreters and scripting languages, and dynamically generating implementations.

The inter-ORB bridge support can also be used to provide interoperability with non-CORBA systems, such as Microsoft's Component Object Model (COM). The ease of doing this will depend on the extent to which those systems conform to the CORBA Object Model.

5.4.4.2 Integrating an ORB in a Gateway

The Common ORB Architecture is designed to allow interoperation with a wide range of object systems (see Figure 5.3). Because there are many existing object systems, a common desire will be to allow the objects in those systems to be accessible via the ORB.

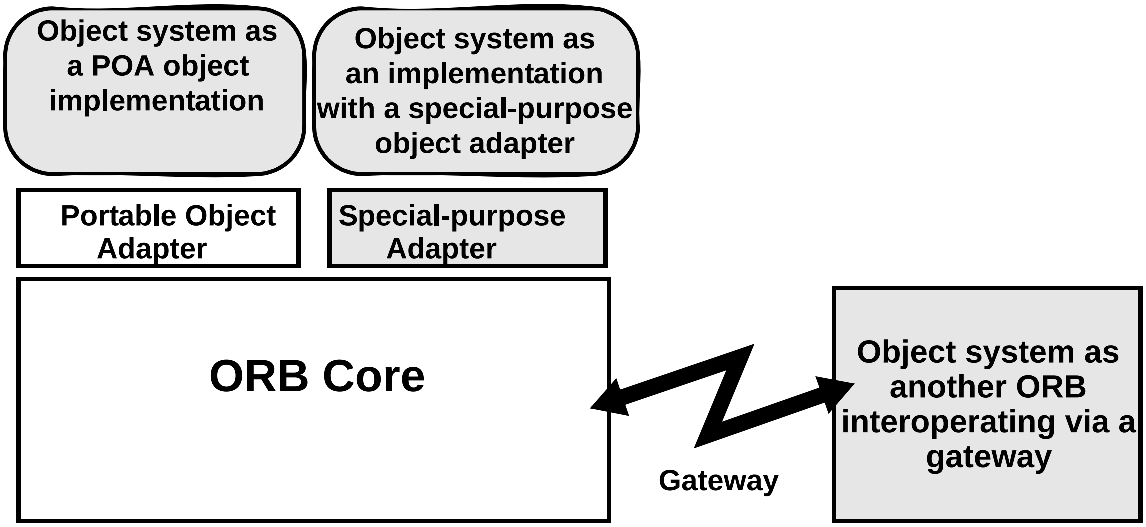
Object system as a POA object implementation
Object system as an implementation with a special-purpose object adapter
Portable Object Adapter
Special-purpose Adapter
ORB Core
Gateway
Object system as another ORB interoperating via a gateway

Figure 5.3- Different Ways to Integrate Foreign Object Systems

For object systems that simply want to map their objects into ORB objects and receive invocations through the ORB, one approach is to have those object systems appear to be implementations of the corresponding ORB objects. The object system would register its objects with the ORB and handle incoming requests, and could act like a client and perform outgoing requests.

In some cases, it will be impractical for another object system to act like a POA object implementation. An object adapter could be designed for objects that are created in conjunction with the ORB and that are primarily invoked through the ORB. Another object system may wish to create objects without consulting the ORB, and might expect most invocations to occur within itself rather than through the ORB. In such a case, a more appropriate object adapter might allow objects to be implicitly registered when they are passed through the ORB.

5.4.5 Quality of Service

5.4.5.1 QoS Abstract Model

The abstract model describes the Quality of Service (QoS) components and their relationships. The specification defines a framework within which current QoS levels are queried and overridden. This framework is intended to be of use for CORBAServices specifiers, as well as for future revisions of CORBA.

5.4.5.2 Model Components

The QoS framework abstract model consists of the following components:

- **Policy** - The base interface from which all QoS objects derive.
- **PolicyList** - A sequence of Policy objects.
- **PolicyManager** - An interface with operations for querying and overriding QoS **Policy** settings.
- **TAG_POLICIES** - A sequence of QoS policy settings contained within an object reference.
- **INVOCATION_POLICIES** - A sequence of QoS policy settings propagated with, and in effect for, an interaction.
- Client-side Policies are applied to control the behavior of requests and replies. For example, Priority, RequestEndTime, and Queueing QoS.
- Server-side Policies are applied to control the default behavior of invocations on a target. For example, QueueOrder and Transactionality QoS.

5.4.5.3 Component Relationships

Programmers set QoS at various levels of scope by creating a Policy object for a particular QoS Policy and selecting the interface for its scope of application. It is anticipated that the following is the standard use-case scenario:

- A POA is created with a certain set of QoS. When object references are created by that POA, the required and supported QoS are encoded in that object reference. Such an object reference is then exported for use by a client.
- Within a client, the ORB's **PolicyManager** interface is obtained to set QoS for the entire ORB (for the entire process when only one ORB is present) either programmatically, or administratively. The Policies set here are valid for all invocations in the process. A programmer-constructed **PolicyList** is used with this interface to actually set the QoS.
- Within that same client, the **CORBA::PolicyCurrent** is obtained to set QoS for all invocations in the current thread. This interface is derived from the **PolicyManager** interface, which can be used to change the QoS for each subsequent invocation performed by that thread. A programmer-constructed **PolicyList** is used with this interface to actually set the QoS.
- Within that same client, the object reference is obtained and an invocation of its **get_client_policy** operation queries the most specific QoS overrides. A programmer-constructed **PolicyList** may be passed to the Object's **set_policy_overrides** operation to obtain a new Object reference with revised QoS. Setting the QoS here applies to all invocations using the new Object reference.
- The effective value for a given policy, the *effective* **Policy**, is determined by arbitrating the policy values for the different scopes. The object reference scope policy supersedes any specified thread (Current) scope policy which, in turn, supersedes any specified ORB scope policy. If no policy value is set at any scope, then the implementation's default behavior will apply.
- The current set of overrides can be validated by calling the Object's pseudo-operation **validate_connection**, which will attempt to locate a target for the object reference if no target has yet been located. At this time, any Policy overrides placed at the Object, Thread, or ORB scope will be reconciled with the QoS Policies established for that object reference when it was created by the POA. The current can then be queried by invoking **get_policy**, which returns the **Policy** value that is in effect.
- Unseen by the application, the ORB (including the protocol engine) modifies its internal behavior in order to realize the quality of service indicated by the client through the steps above.

5.4.5.4 Design Issues

Design decisions were made with respect to the following components of the QoS framework:

- Each QoS is an interface derived from **CORBA::Policy**. The design trade-offs focused on ease of use of application interfaces for setting specific QoS values, extensibility for new QoS types and values, and compactness so the QoS values can be represented efficiently in Service Contexts and IOR Profile Components. Derivation from **CORBA::Policy** allows reuse of existing interfaces and operations for policy management. When certain QoS values must be marshaled in a Service Context or an IOR Profile Component, the most compact format was chosen. The type of QoS **Policy** represented is indicated by a structure containing the integral **PolicyType** and a **sequence** of **octet** holding the values for that policy.

- A generic factory for creating QoS Policies. In the *POA* specification within *CORBA*, each POA **Policy** is created through an operation on the POA itself. Although this presents a convenient typed interface for the creation of **Policy** objects, it causes serious problems when new POA Policies are introduced. To fit with the current model, operations would have to be added to the POA interface for every new type of POA **Policy**. To address this potential administrative nightmare, this specification introduces a new ORB operation **create_policy**. Rather than introducing typed operations for creating all of the Messaging QoS Policies discussed in this specification, the generic factory operation is used.

- A **RebindPolicy** client-side QoS **Policy** to ensure deterministic effective QoS. In *CORBA*, *transparent rebinding* of an object reference may take place during any invocation. Rebinding is defined here to mean changing the client-visible QoS as a result of replacing the IOR Profile used by a client’s object reference with a new IOR Profile. Transparent rebinding is defined as when this happens without notice to the client application. Typically, this happens within GIOP through the use of location forwarding. The default **RebindPolicy** (and the only *CORBA* behavior) supports this transparent rebind. For an application with rigorous quality of service requirements, such transparent rebinding can cause problems. For instance, unexpected errors may occur if the application sets its QoS Policies appropriately for an object reference, and then the ORB transparently changes the application’s assumptions about that reference by obtaining a new IOR. The **RebindPolicy** has been added so that applications can prevent the ORB from silently changing the IOR Profile (and therefore the server-side QoS) that have been assumed. A more rigorous value of this **Policy** even precludes the ORB from silently closing and opening connections (when IIOP is being used, for example). The specific requirements demanded by an application dictate which level of **RebindPolicy** is necessary.

6 OMG IDL Syntax and Semantics

This chapter describes OMG Interface Definition Language (IDL) semantics and gives the syntax for OMG IDL grammatical constructs.

Compliance

Conformant implementations of the CORBA/*e* Compact Profile or CORBA/*e* Micro Profile must comply with all clauses of this chapter; all legal IDL must be accepted. However, certain IDL constructs, even though they must be parsed, may be otherwise ignored. The use of these constructs in the construction of other construction (e.g., as a parameter type for an operation) is illegal and may be rejected by a conforming product. These constructs are enumerated in Table 6.1.

Table 6.1- Constructs Ignored by CORBA/*e* Profiles

Profile	**Ignored Constructs**	**Grammar Rule(s)**
CORBA/*e* Compact	abstract interfaces	6, 7
	value boxes	15
	custom valuetypes	18
	value “supports” interface	19
	context clauses	87, 94
	import	100
CORBA/*e* Micro	abstract interfaces	6, 7
	values	13 through 26, 98
	Any	67
	context clauses	87, 94
	import	100

Additional legality restrictions are contained in later chapters.

6.1 Overview

The OMG Interface Definition Language (IDL) is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation’s parameters. An OMG IDL interface provides the information needed to develop clients that use the interface’s operations.

Clients are not written in OMG IDL, which is purely a descriptive language, but in languages for which mappings from OMG IDL concepts have been defined. The mapping of an OMG IDL concept to a client language construct will depend on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does.

The description of OMG IDL's lexical conventions is presented in Section 6.2, "Lexical Conventions," on page 40. A description of OMG IDL preprocessing is presented in Section 6.3, "Preprocessing," on page 48. The scope rules for identifiers in an OMG IDL specification are described in Section 6.16, "Names and Scoping," on page 87.

OMG IDL is a declarative language. The grammar is presented in Section 6.4, "OMG IDL Grammar," on page 48 and associated semantics is described in the rest of this chapter either in place or through references to other sections of this standard.

OMG IDL-specific pragmas (those not defined for C++) may appear anywhere in a specification; the textual location of these pragmas may be semantically constrained by a particular implementation.

A source file containing interface specifications written in OMG IDL must have an ".idl" extension.

The description of OMG IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). Table 6.2 lists the symbols used in this format and their meaning.

Table 6.2 - IDL EBNF

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times.
+	The preceding syntactic unit can be repeated one or more times.
{}	The enclosed syntactic units are grouped as a single syntactic unit.
[]	The enclosed syntactic unit is optional—may occur zero or one time.

6.2 Lexical Conventions

This section[1] presents the lexical conventions of OMG IDL. It defines tokens in an OMG IDL specification and describes comments, identifiers, keywords, and literals—integer, character, and floating point constants and string literals.

An OMG IDL specification logically consists of one or more files. A file is conceptually translated in several phases.

The first phase is preprocessing, which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space. The result of preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing, is called a translation unit.

[1] This section is an adaptation of *The Annotated C++ Reference Manual,* Chapter 2; it differs in the list of legal keywords and punctuation.

OMG IDL uses the ASCII character set, except for string literals and character literals, which use the ISO Latin-1 (8859.1) character set. The ISO Latin-1 character set is divided into alphabetic characters (letters) digits, graphic characters, the space (blank) character, and formatting characters. Table 6.3 shows the ISO Latin-1 alphabetic characters; upper and lower case equivalences are paired. The ASCII alphabetic characters are shown in the left-hand column of Table 6.3.

Table 6.3- The 114 Alphabetic Characters (Letters)

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	Àà	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	Áá	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	Ââ	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	Ãã	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	Ää	Upper/Lower-case A with diaeresis
Ff	Upper/Lower-case F	Åå	Upper/Lower-case A with ring above
Gg	Upper/Lower-case G	Ææ	Upper/Lower-case dipthong A with E
Hh	Upper/Lower-case H	Çç	Upper/Lower-case C with cedilla
Ii	Upper/Lower-case I	Èè	Upper/Lower-case E with grave accent
Jj	Upper/Lower-case J	Éé	Upper/Lower-case E with acute accent
Kk	Upper/Lower-case K	Êê	Upper/Lower-case E with circumflex accent
Ll	Upper/Lower-case L	Ëë	Upper/Lower-case E with diaeresis
Mm	Upper/Lower-case M	Ìì	Upper/Lower-case I with grave accent
Nn	Upper/Lower-case N	Íí	Upper/Lower-case I with acute accent
Oo	Upper/Lower-case O	Îî	Upper/Lower-case I with circumflex accent
Pp	Upper/Lower-case P	Ïï	Upper/Lower-case I with diaeresis
Qq	Upper/Lower-case Q	Ññ	Upper/Lower-case N with tilde
Rr	Upper/Lower-case R	Òò	Upper/Lower-case O with grave accent
Ss	Upper/Lower-case S	Óó	Upper/Lower-case O with acute accent
Tt	Upper/Lower-case T	Ôô	Upper/Lower-case O with circumflex accent
Uu	Upper/Lower-case U	Õõ	Upper/Lower-case O with tilde
Vv	Upper/Lower-case V	Öö	Upper/Lower-case O with diaeresis
Ww	Upper/Lower-case W	Øø	Upper/Lower-case O with oblique stroke
Xx	Upper/Lower-case X	Ùù	Upper/Lower-case U with grave accent
Yy	Upper/Lower-case Y	Úú	Upper/Lower-case U with acute accent
Zz	Upper/Lower-case Z	Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with diaeresis
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with diaeresis

Table 6.4 lists the decimal digits characters.

Table 6.4- Decimal Digits

0 1 2 3 4 5 6 7 8 9

Table 6.5 shows the graphic characters.

Table 6.5- The 65 Graphic Characters

Char.	Description	Char.	Description
!	exclamation point	¡	inverted exclamation mark
"	double quote	¢	cent sign
#	number sign	£	pound sign
$	dollar sign	¤	currency sign
%	percent sign	¥	yen sign
&	ampersand		broken bar
’	apostrophe	§	section/paragraph sign
(	left parenthesis	¨	diaeresis
)	right parenthesis	©	copyright sign
*	asterisk	ª	feminine ordinal indicator
+	plus sign	«	left angle quotation mark
,	comma	¬	not sign
-	hyphen, minus sign		soft hyphen
.	period, full stop	®	registered trade mark sign
/	solidus	¯	macron
:	colon	°	ring above, degree sign
;	semicolon	±	plus-minus sign
<	less-than sign	^{2}	superscript two
=	equals sign	^{3}	superscript three
>	greater-than sign	´	acute
?	question mark	m	micro
@	commercial at	¶	pilcrow
[	left square bracket	•	middle dot
\	reverse solidus	¸	cedilla
]	right square bracket	^{1}	superscript one
^	circumflex	º	masculine ordinal indicator
_	low line, underscore	»	right angle quotation mark
‘	grave		vulgar fraction 1/4
{	left curly bracket		vulgar fraction 1/2
|	vertical line		vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	¥	multiplication sign
		^{3}	division sign

The formatting characters are shown in Table 6.6.

Table 6.6- The Formatting Characters

Description	Abbreviation	ISO 646 Octal Value
alert	BEL	007
backspace	BS	010
horizontal tab	HT	011
newline	NL, LF	012
vertical tab	VT	013
form feed	FF	014
carriage return	CR	015

6.2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collective, “white space”), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

6.2.2 Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates at the end of the line on which they occur. The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment. Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed, and newline characters.

6.2.3 Identifiers

An identifier is an arbitrarily long sequence of ASCII alphabetic, digit, and underscore (“_”) characters. The first character must be an ASCII alphabetic character. All characters are significant.

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Table 6.3 defines the equivalence mapping of upper- and lower-case letters.
- All characters are significant.

Identifiers that differ only in case collide, and will yield a compilation error under certain circumstances. An identifier for a given definition must be spelled identically (e.g., with respect to case) throughout a specification.

There is only one namespace for OMG IDL identifiers in each scope. Using the same identifier for a constant and an interface, for example, produces a compilation error.

For example:

```
module M {
    typedef long Foo;
    const long thing = 1;
    interface thing {           // error: reuse of identifier
        void doit (
            in Foo foo          // error: Foo and foo collide and refer to different things
        );
        readonly attribute long Attribute; // error: Attribute collides with keyword attribute
    };
};
```

6.2.3.1 Escaped Identifiers

As IDL evolves, new keywords that are added to the IDL language may inadvertently collide with identifiers used in existing IDL and programs that use that IDL. Fixing these collisions will require not only the IDL to be modified, but programming language code that depends upon that IDL will have to change as well. The language mapping rules for the renamed IDL identifiers will cause the mapped identifier names (e.g., method names) to be changed.

To minimize the amount of work, users may lexically "escape" identifiers by prepending an underscore (_) to an identifier. This is a purely lexical convention that ONLY turns off keyword checking. The resulting identifier follows all the other rules for identifier processing. For example, the identifier **_AnIdentifier** is treated as if it were **AnIdentifier**.

The following is a non-exclusive list of implications of these rules:

- The underscore does not appear in the Interface Repository.
- The underscore is not used in the DII and DSI.
- The underscore is not transmitted over "the wire."
- Case sensitivity rules are applied to the identifier after stripping off the leading underscore.

For example:

```
module M {
    interface thing {
        attribute boolean abstract;     // error: abstract collides with
                                        //    keyword abstract
        attribute boolean _abstract;    // ok: abstract is an identifier
    };
};
```

To avoid unnecessary confusion for readers of IDL, it is recommended that interfaces only use the escaped form of identifiers when the unescaped form clashes with a newly introduced IDL keyword. It is also recommended that interface designers avoid defining new identifiers that are known to require escaping. Escaped literals are only recommended for IDL that expresses legacy interface, or for IDL that is mechanically generated.

6.2.4 Keywords

The identifiers listed in Table 6.7 are reserved for use as keywords and may not be used otherwise, unless escaped with a leading underscore.

Table 6.7- Keywords

abstract	exception	inout	provides	truncatable
any	emits	interface	public	typedef
attribute	enum	local	publishes	typeid
boolean	eventtype	long	raises	typeprefix
case	factory	module	readonly	unsigned
char	FALSE	multiple	setraises	union
component	finder	native	sequence	uses
const	fixed	Object	short	ValueBase
consumes	float	octet	string	valuetype
context	getraises	oneway	struct	void
custom	home	out	supports	wchar
default	import	primarykey	switch	wstring
double	in	private	TRUE	

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords (see Section 6.2.3, “Identifiers,” on page 43) are illegal. For example, “**boolean**” is a valid keyword; “**Boolean**” and “**BOOLEAN**” are illegal identifiers.

For example:

```
module M {
    typedef Long Foo;               // Error: keyword is long not Long
    typedef boolean BOOLEAN;        // Error: BOOLEAN collides with
                                    // the keyword boolean;
};
```

OMG IDL specifications use the characters shown in Table 6.8 as punctuation.

Table 6.8- Punctuation Characters

;	{	}	:	,	=	+	-	(	)	<	>	[	]
'	"	\	|	^	&	*	/	%	~				

In addition, the tokens listed in Table 6.9 are used by the preprocessor.

Table 6.9- Preprocessor Tokens

#	##	!	||	&&

6.2.5 Literals

This section describes the following literals:

- Integer

- Character
- Floating-point
- String
- Fixed-point

6.2.5.1 Integer Literals

An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero). A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. A sequence of digits preceded by 0x or 0X is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include a or A through f or F with decimal values ten through fifteen, respectively. For example, the number twelve can be written 12, 014, or 0XC.

6.2.5.2 Character Literals

A character literal is one or more characters enclosed in single quotes, as in ’x.’ Character literals have type **char**.

A character is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit, or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859.1) character set standard (see Table 6.3, Table 6.4, and Table 6.5). The value of a null is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard (see Table 6.6). The meaning of all other characters is implementation-dependent.

Non-graphic characters must be represented using escape sequences as defined below in Table 6.10. Note that escape sequences must be used to represent single quote and backslash characters in character literals.

Table 6.10- Escape Sequences

Description	Escape Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"
octal number	\ooo
hexadecimal number	\xhh
unicode character	\uhhhh

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhh consists of the backslash followed by x followed by one or two hexadecimal digits that are taken to specify the value of the desired character.

The escape \uhhhh consists of a backslash followed by the character ‘u,’ followed by one, two, three, or four hexadecimal digits. This represents a unicode character literal. Thus the literal “\u002E” represents the unicode period ‘.’ character and the literal “\u3BC” represents the unicode greek small letter ‘mu.’ The \u escape is valid only with wchar and wstring types. Because a wide string literal is defined as a sequence of wide character literals a sequence of \u literals can be used to define a wide string literal. Attempts to set a char type to a \u defined literal or a string type to a sequence of \u literals result in an error.

A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest char.

Wide character literals have an **L** prefix, for example:

```
const wchar C1 = L'X';
```

Attempts to assign a wide character literal to a non-wide character constant or to assign a non-wide character literal to a wide character constant result in a compile-time diagnostic.

Both wide and non-wide character literals must be specified using characters from the ISO 8859-1 character set.

6.2.5.3 Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter e (or E) and the exponent (but not both) may be missing.

6.2.5.4 String Literals

A string literal is a sequence of characters (as defined in Section 6.2.5.2, “Character Literals,” on page 46), with the exception of the character with numeric value 0, surrounded by double quotes, as in “...”.

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example,

```
"\xA" "B"
```

contains the two characters '\xA' and 'B' after concatenation (and not the single hexadecimal character '\xAB').

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. Within a string, the double quote character **"** must be preceded by a \.

A string literal may not contain the character ‘\0.’

Wide string literals have an L prefix, for example:

```
const wstring S1 = L"Hello";
```

Attempts to assign a wide string literal to a non-wide string constant or to assign a non-wide string literal to a wide string constant result in a compile-time diagnostic.

Both wide and non-wide string literals must be specified using characters from the ISO 8859-1 character set.

A wide string literal shall not contain the wide character with value zero.

6.2.5.5 Fixed-Point Literals

A fixed-point decimal literal consists of an integer part, a decimal point, a fraction part, and a d or D. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. Either the integer part or the fraction part (but not both) may be missing; the decimal point (but not the letter d (or D)) may be missing.

6.3 Preprocessing

OMG IDL is preprocessed according to the specification of the preprocessor in “International Organization for Standardization. 1998. ISO/IEC 14882 Standard for the C++ Programming Language. Geneva: International Organization for Standardization.” The preprocessor may be implemented as a separate process or built into the IDL compiler.

Lines beginning with # (also called “directives”) communicate with this preprocessor. White space may appear before the #. These lines have syntax independent of the rest of OMG IDL; they may appear anywhere and have effects that last (independent of the OMG IDL scoping rules) until the end of the translation unit. The textual location of OMG IDL-specific pragmas may be semantically constrained.

A preprocessing directive (or any line) may be continued on the next line in a source file by placing a backslash character (“\”), immediately before the newline at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the newline before the input sequence is divided into tokens. A backslash character may not be the last character in a source file.

A preprocessing token is an OMG IDL token (see Section 6.2.1, “Tokens,” on page 43), a file name as in a **#include** directive, or any single character other than white space that does not match another preprocessing token.

The primary use of the preprocessing facilities is to include definitions from other OMG IDL specifications. Text in files included with a **#include** directive is treated as if it appeared in the including file, except that **RepositoryId** related pragmas are handled in a special way. The special handling of these pragmas is described in Section 4.3, “Pragma Directives for RepositoryId,” on page 3.

Note that whether a particular IDL compiler generates code for included files is an implementation-specific issue. To support separate compilation, IDL compilers may not generate code for included files, or do so only if explicitly instructed.

6.4 OMG IDL Grammar

```
(1)    <specification>  ::=  <import>* <definition>+
(2)       <definition>  ::=  <type_dcl> “;”
                         |   <const_dcl> “;”
                         |   <except_dcl> “;”
                         |   <interface> “;”
                         |   <module> “;”
                         |   <value> “;”
                         |   <type_id_dcl> “;”
                         |   <type_prefix_dcl> “;”
(3)           <module>  ::=  “module” <identifier> “{“ <definition>+ “}”
```

```
(4)              <interface>   ::=  <interface_dcl>
                               |    <forward_dcl>
(5)          <interface_dcl>   ::=  <interface_header> “{” <interface_body> “}”
(6)            <forward_dcl>   ::=  [ “abstract” | “local” ] “interface” <identifier>
(7)       <interface_header>   ::=  [ “abstract” | “local” ] “interface” <identifier>
                               [ <interface_inheritance_spec> ]
(8)         <interface_body>   ::=  <export>*
(9)                 <export>   ::=  <type_dcl> “;”
                               |    <const_dcl> “;”
                               |    <except_dcl> “;”
                               |    <attr_dcl> “;”
                               |    <op_dcl> “;”
                               |    <type_id_dcl> “;”
                               |    <type_prefix_dcl> “;”
(10)<interface_inheritance_spec>::=“:” <interface_name>
                               { “,” <interface_name> }*
(11)      <interface_name>   ::=  <scoped_name>
(12)         <scoped_name>   ::=  <identifier>
                               |    “::” <identifier>
                               |    <scoped_name> “::” <identifier>
(13)               <value>   ::=  ( <value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
(14)<value_forward_dcl>   ::=  [ “abstract” ] “valuetype” <identifier>
(15)       <value_box_dcl>   ::=  “valuetype”  <identifier> <type_spec>
(16)       <value_abs_dcl>   ::=  “abstract” “valuetype” <identifier>
                               [ <value_inheritance_spec> ]
                               “{“ <export>* “}”
(17)           <value_dcl>   ::=  <value_header> “{“  < value_element>* “}”
(18)        <value_header>   ::=  [“custom” ] “valuetype” <identifier>
                               [ <value_inheritance_spec> ]
(19)<value_inheritance_spec>::=[ “:” [ “truncatable” ] <value_name>
                               { “,”  <value_name> }* ]
                               [ “supports” <interface_name>
                               { “,” <interface_name> }*  ]
(20)          <value_name>   ::=  <scoped_name>
(21)       <value_element>   ::=  <export> | < state_member> | <init_dcl>
(22)        <state_member>   ::=  ( “public” | “private” )
                               <type_spec> <declarators> “;”
(23)            <init_dcl>   ::=  “factory” <identifier>
                               “(“  [ <init_param_decls> ]  “)”
                               [ <raises_expr> ] “;”
(24)    <init_param_decls>   ::=  <init_param_decl> { “,” <init_param_decl> }*
(25)     <init_param_decl>   ::=  <init_param_attribute> <param_type_spec> <simple_declarator>
(26)<init_param_attribute>::=  “in”
(27)           <const_dcl>   ::=  “const” <const_type>
                               <identifier> “=” <const_exp>
(28)          <const_type>   ::=  <integer_type>
                               |    <char_type>
```

```
                         |   <wide_char_type>
                         |   <boolean_type>
                         |   <floating_pt_type>
                         |   <string_type>
                         |   <wide_string_type>
                         |   <fixed_pt_const_type>
                         |   <scoped_name>
                         |   <octet_type>
(29)         <const_exp> ::= <or_expr>
(30)           <or_expr> ::= <xor_expr>
                         |   <or_expr> “|” <xor_expr>
(31)          <xor_expr> ::= <and_expr>
                         |   <xor_expr> “^” <and_expr>
(32)          <and_expr> ::= <shift_expr>
                         |   <and_expr> “&” <shift_expr>
(33)        <shift_expr> ::= <add_expr>
                         |   <shift_expr> “>>” <add_expr>
                         |   <shift_expr> “<<” <add_expr>
(34)          <add_expr> ::= <mult_expr>
                         |   <add_expr> “+” <mult_expr>
                         |   <add_expr> “-” <mult_expr>
(35)         <mult_expr> ::= <unary_expr>
                         |   <mult_expr> “*” <unary_expr>
                         |   <mult_expr> “/” <unary_expr>
                         |   <mult_expr> “%” <unary_expr>
(36)        <unary_expr> ::= <unary_operator> <primary_expr>
                         |   <primary_expr>
(37)    <unary_operator> ::= “-”
                         |   “+”
                         |   “~”
(38)      <primary_expr> ::= <scoped_name>
                         |   <literal>
                         |   “(” <const_exp> “)”
(39)           <literal> ::= <integer_literal>
                         |   <string_literal>
                         |   <wide_string_literal>
                         |   <character_literal>
                         |   <wide_character_literal>
                         |   <fixed_pt_literal>
                         |   <floating_pt_literal>
                         |   <boolean_literal>
(40)   <boolean_literal> ::= “TRUE”
                         |   “FALSE”
(41)<positive_int_const> ::= <const_exp>
(42)          <type_dcl> ::= “typedef” <type_declarator>
                         |   <struct_type>
                         |   <union_type>
                         |   <enum_type>
```

```
                                |    “native” <simple_declarator>
                                |    <constr_forward_decl>
(43)      <type_declarator>   ::=  <type_spec> <declarators>
(44)            <type_spec>   ::=  <simple_type_spec>
                                |    <constr_type_spec>
(45) <simple_type_spec>   ::=  <base_type_spec>
                                |    <template_type_spec>
                                |    <scoped_name>
(46)     <base_type_spec>   ::=  <floating_pt_type>
                                |    <integer_type>
                                |    <char_type>
                                |    <wide_char_type>
                                |    <boolean_type>
                                |    <octet_type>
                                |    <any_type>
                                |    <object_type>
                                |    <value_base_type>
(47)<template_type_spec>::=  <sequence_type>
                                |    <string_type>
                                |    <wide_string_type>
                                |    <fixed_pt_type>
(48) <constr_type_spec>   ::=  <struct_type>
                                |    <union_type>
                                |    <enum_type>
(49)          <declarators>   ::=  <declarator> { “,” <declarator> }*
(50)           <declarator>   ::=  <simple_declarator>
                                |    <complex_declarator>
(51) <simple_declarator>   ::=  <identifier>
(52)<complex_declarator> ::=  <array_declarator>
(53)    <floating_pt_type>   ::=  “float”
                                |    “double”
                                |    “long” “double”
(54)         <integer_type>   ::=  <signed_int>
                                |    <unsigned_int>
(55)           <signed_int>   ::=  <signed_short_int>
                                |    <signed_long_int>
                                |    <signed_longlong_int>
(56)  <signed_short_int>   ::=  “short”
(57)    <signed_long_int>   ::=  “long”
(58)<signed_longlong_int>::=  “long” “long”
(59)        <unsigned_int>   ::=  <unsigned_short_int>
                                |    <unsigned_long_int>
                                |    <unsigned_longlong_int>
(60)<unsigned_short_int> ::=  “unsigned” “short”
(61)<unsigned_long_int>   ::=  “unsigned” “long”
(62)<unsigned_longlong_int>::=“unsigned” “long” “long”
(63)            <char_type>   ::=  “char”
```

```
(64)     <wide_char_type>  ::=  “wchar”
(65)       <boolean_type>  ::=  “boolean”
(66)         <octet_type>  ::=  “octet”
(67)           <any_type>  ::=  “any”
(68)        <object_type>  ::=  “Object”
(69)        <struct_type>  ::=  “struct” <identifier> “{” <member_list> “}”
(70)        <member_list>  ::=  <member>+
(71)             <member>  ::=  <type_spec> <declarators> “;”
(72)         <union_type>  ::=  “union” <identifier> “switch”
                               “(” <switch_type_spec> “)”
                               “{” <switch_body> “}”
(73) <switch_type_spec>    ::=  <integer_type>
                               |    <char_type>
                               |    <boolean_type>
                               |    <enum_type>
                               |    <scoped_name>
(74)        <switch_body>  ::=  <case>+
(75)               <case>  ::=  <case_label>+ <element_spec> “;”
(76)         <case_label>  ::=  “case” <const_exp> “:”
                               |    “default” “:”
(77)       <element_spec>  ::=  <type_spec> <declarator>
(78)          <enum_type>  ::=  “enum” <identifier>
                               “{” <enumerator> { “,” <enumerator> }* “}”
(79)         <enumerator>  ::=  <identifier>
(80)      <sequence_type>  ::=  “sequence” “<” <simple_type_spec> “,” <positive_int_const> “>”
                               |    “sequence” “<” <simple_type_spec> “>”
(81)        <string_type>  ::=  “string” “<” <positive_int_const> “>”
                               |    “string”
(82)  <wide_string_type>   ::=  “wstring” “<” <positive_int_const> “>”
                               |    “wstring”
(83)   <array_declarator>  ::=  <identifier> <fixed_array_size>+
(84)   <fixed_array_size>  ::=  “[” <positive_int_const> “]”
(85)           <attr_dcl>  ::=   <readonly_attr_spec>
                               |     <attr_spec>
(86)         <except_dcl>  ::=  “exception” <identifier> “{“ <member>* “}”
(87)             <op_dcl>  ::=  [ <op_attribute> ] <op_type_spec>
                               <identifier> <parameter_dcls>
                               [ <raises_expr> ] [ <context_expr> ]
(88)       <op_attribute>  ::=  “oneway”
(89)       <op_type_spec>  ::=  <param_type_spec>
                               |    “void”
(90)     <parameter_dcls>  ::=  “(” <param_dcl> { “,” <param_dcl> }* “)”
                               |    “(” “)”
(91)          <param_dcl>  ::=  <param_attribute> <param_type_spec> <simple_declarator>
(92)    <param_attribute>  ::=  “in”
                               |    “out”
```

```
                                |   "inout"
(93)        <raises_expr>   ::=  "raises" "(" <scoped_name>
                                { "," <scoped_name> }* ")"
(94)       <context_expr>   ::=  "context" "(" <string_literal>
                                { "," <string_literal> }* ")"
(95) <param_type_spec>   ::=  <base_type_spec>
                                |   <string_type>
                                |   <wide_string_type>
                                |   <scoped_name>
(96)       <fixed_pt_type>   ::=  "fixed" "<" <positive_int_const> "," <positive_int_const> ">"
(97)<fixed_pt_const_type>::=  "fixed"
(98)  <value_base_type>   ::=  "ValueBase"
(99)<constr_forward_decl>::= "struct" <identifier>
                                |   "union" <identifier>
(100)              <import>   ::=  "import" <imported_scope> ";"
(101)  <imported_scope>   ::=   <scoped_name> | <string_literal>
(102)        <type_id_dcl>   ::=  "typeid" <scoped_name> <string_literal>
(103)   <type_prefix_dcl>   ::=  "typeprefix" <scoped_name> <string_literal>
(104)<readonly_attr_spec> ::=  "readonly" "attribute" <param_type_spec>  <readonly_attr_declarator>
(105)<readonly_attr_declarator>::=  <simple_declarator> <raises_expr>
                                |   <simple_declarator>
                                    { "," <simple_declarator> }*
(106)           <attr_spec>    ::= "attribute" <param_type_spec> <attr_declarator>
(107)    <attr_declarator>   ::=  <simple_declarator> <attr_raises_expr>
                                |   <simple_declarator>
                                     { "," <simple_declarator> }*
(108)  <attr_raises_expr>   ::=  <get_excep_expr> [ <set_excep_expr> ]
                                |   <set_excep_expr>
(109)  <get_excep_expr>   ::=  "getraises" <exception_list>
(110)   <set_excep_expr>   ::=  "setraises" <exception_list>
(111)     <exception_list>   ::=  "(" <scoped_name>
                                     { "," <scoped_name> } * ")"
```

6.5 OMG IDL Specification

An OMG IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions. The syntax is:

```
(1)          <specification>   ::=  <import>* <definition>+
(2)               <definition>   ::=  <type_dcl> ";"
                                |   <const_dcl> ";"
                                |   <except_dcl> ";"
                                |   <interface> ";"
                                |   <module> ";"
                                |   <value> ";"
                                |   <type_id_dcl> ";"
                                |   <type_prefix_dcl> ";"
```

See Section 6.6, “Module Declaration,” on page 54, for the specification of <module>.

See Section 6.7, “Interface Declaration,” on page 54, for the specification of <interface>.

See Section 6.8, “Value Declaration,” on page 59, for the specification of <value>.

See Section 6.9, “Constant Declaration,” on page 64, Section 6.10, “Type Declaration,” on page 68, and Section 6.11, “Exception Declaration,” on page 80 respectively for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

See Section 6.14, “Repository Identity Related Declarations,” on page 84, for specification of Repository Identity declarations which include <type_id_dcl> and <type_prefix_dcl>.

6.6 Module Declaration

A module definition satisfies the following syntax:

(3)	**<module>**	**::=**	**“module” <identifier> “{“ <definition>$^+$ “}”**

The module construct is used to scope OMG IDL identifiers; see Section 6.15, “CORBA Module,” on page 86 for details.

6.7 Interface Declaration

An interface definition satisfies the following syntax:

(4)	**<interface>**	**::=**	**<interface_dcl>**
		|	**<forward_dcl>**
(5)	**<interface_dcl>**	**::=**	**<interface_header> “{” <interface_body> “}”**
(6)	**<forward_dcl>**	**::=**	**[“abstract” | “local”] “interface” <identifier>**
(7)	**<interface_header>**	**::=**	**[“abstract” | “local”] “interface” <identifier>** **[<interface_inheritance_spec>]**
(8)	**<interface_body>**	**::=**	**<export>***
(9)	**<export>**	**::=**	**<type_dcl> “;”**
		|	**<const_dcl> “;”**
		|	**<except_dcl> “;”**
		|	**<attr_dcl> “;”**
		|	**<op_dcl> “;”**
		|	**<type_id_decl> “;”**
		|	**<type_prefix_decl> “;”**

6.7.1 Interface Header

The interface header consists of three elements:

1. An optional modifier specifying if the interface is an abstract interface.
2. The interface name. The name must be preceded by the keyword ***interface***, and consists of an identifier that names the interface.
3. An optional inheritance specification. The inheritance specification is described in the next section.

The **<identifier>** that names an interface defines a legal type name. Such a type name may be used anywhere an **<identifier>** is legal in the grammar, subject to semantic constraints as described in the following sections. Since one can only hold references to an object, the meaning of a parameter or structure member, which is an interface type is as a *reference* to an object supporting that interface. Each language binding describes how the programmer must represent such interface references.

Abstract interfaces have slightly different rules and semantics from “regular” interfaces, as described in Section 6.7.6, “Abstract Interface,” on page 59. They also follow different language mapping rules.

Local interfaces have slightly different rules and semantics from “regular” interfaces, as described in Section 6.7.7, “Local Interface,” on page 59. They also follow different language mapping rules.

6.7.2 Interface Inheritance Specification

The syntax for inheritance is as follows:

```
(10)    <interface_inheritance_spec>::=“:” <interface_name>
                                   { “,” <interface_name> }*
(11)   <interface_name>   ::=  <scoped_name>
(12)     <scoped_name>    ::=  <identifier>
                           |   “::” <identifier>
                           |   <scoped_name> “::” <identifier>
```

Each **<scoped_name>** in an **<interface_inheritance_spec>** must be the name of a previously defined interface or an alias to a previously defined interface. See Section 6.7.5, “Interface Inheritance,” on page 56 for the description of inheritance.

6.7.3 Interface Body

The interface body contains the following kinds of declarations:

- Constant declarations, which specify the constants that the interface exports; constant declaration syntax is described in Section 6.9, “Constant Declaration,” on page 64.
- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in Section 6.10, “Type Declaration,” on page 68.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in Section 6.11, “Exception Declaration,” on page 80.
- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in Section 6.13, “Attribute Declaration,” on page 83.
- Operation declarations, which specify the operations that the interface exports and the format of each, including operation name, the type of data returned, the types of all parameters of an operation, legal exceptions that may be returned as a result of an invocation, and contextual information that may affect method dispatch; operation declaration syntax is described in Section 6.12, “Operation Declaration,” on page 80.

Empty interfaces are permitted (that is, those containing no declarations).

Some implementations may require interface-specific pragmas to precede the interface body.

6.7.4 Forward Declaration

A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other. The syntax is: optionally either the keyword **abstract** or the keyword **local**, followed by the keyword **interface**, followed by an <identifier> that names the interface.

Multiple forward declarations of the same interface name are legal.

It is illegal to inherit from a forward-declared interface whose definition has not yet been seen:

```
module Example {
    interface base;                  // Forward declaration

    // ...

    interface derived : base {};     // Error
    interface base {};               // Define base
    interface derived : base {};     // OK
};
```

6.7.5 Interface Inheritance

An interface can be derived from another interface, which is then called a *base* interface of the derived interface. A derived interface, like all interfaces, may declare new elements (constants, types, attributes, exceptions, and operations). In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator ("::") may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any of the type, constant, and exception names that have been inherited; the scope rules for such names are described in Section 6.16, "Names and Scoping," on page 87.

An interface is called a direct base if it is mentioned in the **<interface_inheritance_spec>** and an indirect base if it is not a direct base but is a base interface of one of the interfaces mentioned in the **<interface_inheritance_spec>**.

An interface may be derived from any number of base interfaces. Such use of more than one direct base interface is often called multiple inheritance. The order of derivation is not significant.

An abstract interface may only inherit from other abstract interfaces.

An interface may not be specified as a direct base interface of a derived interface more than once; it may be an indirect base interface more than once. Consider the following example:

```
interface A { ... }
interface B: A { ... }
interface C: A { ... }
interface D: B, C { ... }

interface E: A, B { ... };                        // OK
```

The relationships between these interfaces is shown in Figure 6.1. This “diamond” shape is legal, as is the definition of E on the right.

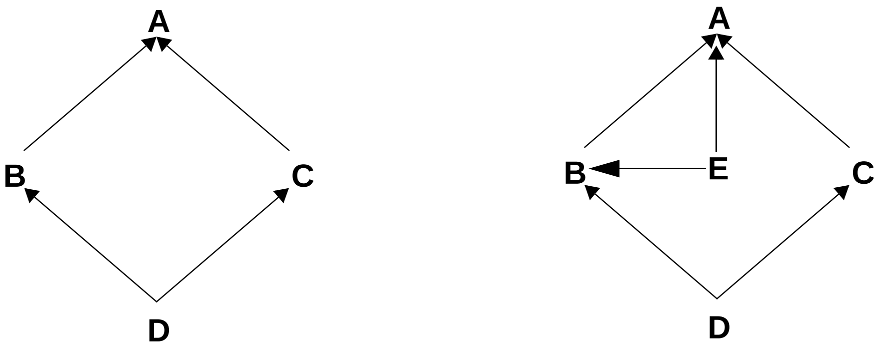

Figure 6.1- Legal Multiple Inheritance Example

References to base interface elements must be unambiguous. A Reference to a base interface element is ambiguous if the name is declared as a constant, type, or exception in more than one base interface. Ambiguities can be resolved by qualifying a name with its interface name (that is, using a **<scoped_name>**). It is illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.

So for example in:

```
interface A {
    typedef long L1;
    short opA(in L1 l_1);
};

interface B {
    typedef short L1;
    L1 opB(in long l);
};

interface C: B, A {
    typedef L1 L2;              // Error: L1 ambiguous
    typedef A::L1 L3;           // A::L1 is OK
    B::L1 opC(in L3 l_3);       // all OK no ambiguities
};
```

References to constants, types, and exceptions are bound to an interface when it is defined (i.e., replaced with the equivalent global **<scoped_name>**s). This guarantees that the syntax and semantics of an interface are not changed when the interface is a base interface for a derived interface. Consider the following example:

```
const long L = 3;

interface A {
    typedef float coord[L]:
    void f (in coord s);            // s has three floats
};
```

```
interface B {
    const long L = 4;
};

interface C: B, A { };                  // what is C::f()'s signature?
```

The early binding of constants, types, and exceptions at interface definition guarantees that the signature of operation f in interface C is

```
typedef float coord[3];
void f (in coord s);
```

which is identical to that in interface A. This rule also prevents redefinition of a constant, type, or exception in the derived interface from affecting the operations and attributes inherited from a base interface.

Interface inheritance causes all identifiers defined in base interfaces, both direct and indirect, to be visible in the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope can be redefined in the current scope. An attempt to use an ambiguous name without qualification produces a compilation error. Thus in:

```
interface A {
    typedef string<128> string_t;
};

interface B {
    typedef string<256> string_t;
};

interface C: A, B {
    attribute string_t       Title;        // Error: string_t ambiguous
    attribute A::string_t    Name;         // OK
    attribute B::string_t    City;         // OK
};
```

operation and attribute names are used at run-time by both the stub and dynamic interfaces. As a result, all operations and attributes that might apply to a particular object must have unique names. This requirement prohibits redefining an operation or attribute name in a derived interface, as well as inheriting two operations or attributes with the same name.

```
interface A {
    void make_it_so();
};

interface B: A {
    short make_it_so(in long times);   // Error: redefinition of make_it_so
};
```

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 6.11 on page 64.

6.7.5.1 Abstract Interface

An interface declaration containing the keyword **abstract** in its header, declares an abstract interface. The following special rules apply to abstract interfaces:

- Abstract interfaces may only inherit from other abstract interfaces.
- Value types may support any number of abstract interfaces.

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 6.11 on page 64.

6.7.6 Local Interface

An interface declaration containing the keyword **local** in its header, declares a local interface. An interface declaration not containing the keyword **local** is referred to as an unconstrained interface. An object implementing a local interface is referred to as a local object. The following special rules apply to local interfaces:

- A local interface may inherit from other local or unconstrained interfaces.
- An unconstrained interface may not inherit from a local interface. An interface derived from a local interface must be explicitly declared local.
- A valuetype may support a local interface.
- Any IDL type, including an unconstrained interface, may appear as a parameter, attribute, return type, or exception declaration of a local interface.
- A local interface is a local type, as is any non-interface type declaration constructed using a local interface or other local type. For example, a struct, union, or exception with a member that is a local interface is also itself a local type.
- A local type may be used as a parameter, attribute, return type, or exception declaration of a local interface or of a valuetype.
- A local type may not appear as a parameter, attribute, return type, or exception declaration of an unconstrained interface.

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 6.11 on page 64.

See Section 6.2.12, “LocalObject Operations,” on page 10 for CORBA implementation semantics associated with local objects.

6.8 Value Declaration

There are several kinds of value type declarations: “regular” value types, boxed value types, abstract value types, and forward declarations.

A value declaration satisfies the following syntax:

(13) <value> ::= (<value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)

6.8.1 Regular Value Type

A regular value type satisfies the following syntax:

```
(17)          <value_dcl>   ::=  <value_header> “{“  < value_element>* “}”
(18)       <value_header>   ::=  [“custom” ] “valuetype” <identifier>
                                 [ <value_inheritance_spec> ]
(21)      <value_element>   ::=  <export>
                                 |     < state_member> |
                                 |     <init_dcl>
```

6.8.1.1 Value Header

The value header consists of two elements:

1. The value type’s name and optional modifier specifying whether the value type uses custom marshaling.
2. An optional value inheritance specification. The value inheritance specification is described in the next section.

6.8.1.2 Value Element

A value can contain all the elements that an interface can as well as the definition of state members, and initializers for that state.

6.8.1.3 Value Inheritance Specification

```
(19)<value_inheritance_spec>::=[ “:” [ “truncatable” ] <value_name>
                                 { “,”  <value_name> }* ]
                                 [ “supports” <interface_name>
                                 { “,” <interface_name> }*  ]
(20)         <value_name>   ::=  <scoped_name>
```

Each **<value_name>** in a **<value_inheritance_spec>** must be the name of a previously defined value type or an alias to a previously defined value type. Each **<interface_name>** in a **<value_inheritance_spec>** must be the name of a previously defined interface or an alias to a previously defined interface. See Section 6.8.5, “Valuetype Inheritance,” on page 63 for the description of value type inheritance.

The **truncatable** modifier may not be used if the value type being defined is a custom value.

A valuetype that supports a local interface does not itself become *local* (i.e., unmarshalable) as a result of that support.

6.8.1.4 State Members

```
(22)       <state_member>   ::=  ( “public” | “private” )
                                 <type_spec> <declarators> “;”
```

Each **<state_member>** defines an element of the state, which is marshaled and sent to the receiver when the value type is passed as a parameter. A state member is either public or private. The annotation directs the language mapping to hide or expose the different parts of the state to the clients of the value type. The private part of the state is only accessible to the implementation code and the marshaling routines.

A valuetype that has a state member that is *local* (i.e., non-marshalable like a local interface), is itself rendered *local*. That is, such valuetypes behave similar to local interfaces when an attempt is made to marshal them.

Note that certain programming languages may not have the built in facilities needed to distinguish between the public and private members. In these cases, the language mapping specifies the rules that programmers are responsible for following.

6.8.1.5 Initializers

```
(23)               <init_dcl>   ::=  "factory" <identifier>
                                     "("  [ <init_param_decls> ]  ")"
                                     [ <raises_expr> ] ";"
(24)  <init_param_decls>   ::=  <init_param_decl> { "," <init_param_decl> }*
(25)    <init_param_decl>   ::=  <init_param_attribute> <param_type_spec> <simple_declarator>
(26)<init_param_attribute>::=  "in"
```

In order to ensure portability of value implementations, designers may also define the signatures of initializers (or constructors) for non-abstract value types. Syntactically these look like local operation signatures except that they are prefixed with the keyword **factory**, have no return type, and must use only in parameters. There may be any number of factory declarations. The names of the initializers are part of the name scope of the value type. Initializers defined in a valuetype are not inherited by derived valuetypes, and hence the names of the initializers are free to be reused in a derived valuetype.

If no initializers are specified in IDL, the value type does not provide a portable way of creating a runtime instance of its type. There is no default initializer. This allows the definition of IDL value types, which are not intended to be directly instantiated by client code.

6.8.1.6 Value Type Example

```
interface Tree {
    void print()
};

valuetype WeightedBinaryTree {
    // state definition
        private unsigned long weight;
        private WeightedBinaryTree left;
        private WeightedBinaryTree right;
    // initializer
        factory init(in unsigned long w);
    // local operations
        WeightSeq pre_order();
        WeightSeq post_order();
};
valuetype WTree: WeightedBinaryTree supports Tree {};
```

6.8.2 Boxed Value Type

```
(15)       <value_box_dcl>   ::=  "valuetype"  <identifier> <type_spec>
```

It is often convenient to define a value type with no inheritance or operations and with a single state member. A shorthand IDL notation is used to simplify the use of value types for this kind of simple containment, referred to as a “value box.”

Since a value box of a valuetype adds no additional properties to a valuetype, it is an error to box valuetypes.

Value box is particularly useful for strings and sequences. Basically one does not have to create what is in effect an additional namespace that will contain only one name.

An example is the following IDL:

```
module Example {
    interface Foo {
        ...  /* anything */
    };
    valuetype FooSeq sequence<Foo>;
    interface Bar {
        void doIt (in FooSeq seq1);
    };
};
```

The above IDL provides similar functionality to writing the following IDL. However the type identities (repository ID’s) would be different.

```
module Example {
    interface Foo {
        ...  /* anything */
    };
    valuetype FooSeq {
        public sequence<Foo> data;
    };
    interface Bar {
        void doIt (in FooSeq seq);
    };
};
```

The former is easier to manipulate after it is mapped to a concrete programming language.

Any IDL type may be used to declare a value box except for a valuetype.

The declaration of a boxed value type does not open a new scope. Thus a construction such as:

```
    valuetype FooSeq sequence <FooSeq>;
```

is not legal IDL. The identifier being declared as a boxed value type cannot be used subsequent to its initial use and prior to the completion of the boxed value declaration.

6.8.3 Abstract Value Type

```
(16)    <value_abs_dcl>  ::=  “abstract” “valuetype” <identifier>
                              [ <value_inheritance_spec> ]
                              “{“ <export>* “}”
```

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. No <state_member> or <initializers> may be specified. However, local operations may be specified. Essentially they are a bundle of operation signatures with a purely local implementation.

Note that a concrete value type with an empty state is not an abstract value type.

6.8.4 Value Forward Declaration

(14)<value_forward_dcl> ::= ["abstract"] "valuetype" <identifier>

A forward declaration declares the name of a value type without defining it. This permits the definition of value types that refer to each other. The syntax consists simply of the keyword **valuetype** followed by an **<identifier>** that names the value type.

Multiple forward declarations of the same value type name are legal.

Boxed value types cannot be forward declared; such a forward declaration would refer to a normal value type.

It is illegal to inherit from a forward-declared value type whose definition has not yet been seen.

It is illegal for a value type to support a forward-declared interface whose definition has not yet been seen.

6.8.5 Valuetype Inheritance

The terminology that is used to describe value type inheritance is directly analogous to that used to describe interface inheritance (see Section 6.7.5, "Interface Inheritance," on page 56).

The name scoping and name collision rules for valuetypes are identical to those for interfaces. In addition, no valuetype may be specified as a direct abstract base of a derived valuetype more than once; it may be an indirect abstract base more than once. See Section 6.7.5, "Interface Inheritance," on page 56 for a detailed description of the analogous properties for interfaces.

Values may be derived from other values and can support an interface and any number of abstract interfaces.

Once implementation (state) is specified at a particular point in the inheritance hierarchy, all derived value types (which must of course implement the state) may only derive from a single (concrete) value type. They can however derive from other additional abstract values and support an additional interface.

The single immediate base concrete value type, if present, must be the first element specified in the inheritance list of the value declaration's IDL. It may be followed by other abstract values from which it inherits. The interface and abstract interfaces that it supports are listed following the **supports** keyword.

While a valuetype may only directly support one interface, it is possible for the valuetype to support other interfaces as well through inheritance. In this case, the supported interface must be derived, directly or indirectly, from each interface that the valuetype supports through inheritance. This rule does not apply to abstract interfaces that the valuetype supports. For example:

```
interface I1 { };
interface I2 { };
interface I3: I1, I2 { };

abstract valuetype V1 supports I1 { };
abstract valuetype V2 supports I2 { };
```

```
valuetype V3: V1, V2 supports I3 { }; // legal
valuetype V4: V1 supports I2 { }; // illegal
```

A stateful value that derives from another stateful value may specify that it is **truncatable**. This means that it is to "truncate" an instance to be an instance of any of its truncatable parent (stateful) value types under certain conditions. Note that all the intervening types in the inheritance hierarchy must be truncatable in order for truncation to a particular type to be allowed.

Because custom values require an exact type match between the sending and receiving context, **truncatable** may not be specified for a custom value type.

Non-custom value types may not (transitively) inherit from custom value types.

Boxed value types may not be derived from, nor may they derive from anything else.

These rules are summarized in the following table:

Table 6.11- Allowable Inheritance Relationships

May inherit from:	Interface	Abstract Interface	Abstract Value	Stateful Value	Boxed value
Interface	multiple	multiple	no	no	no
Abstract Interface	no	multiple	no	no	no
Abstract Value	supports single	supports multiple	multiple	no	no
Stateful Value	supports single	supports multiple	multiple	single (may be truncatable)	no
Boxed Value	no	no	no	no	no

6.9 Constant Declaration

This section describes the syntax for constant declarations.

6.9.1 Syntax

The syntax for a constant declaration is:

```
(27)          <const_dcl>  ::=  "const" <const_type>
                                <identifier> "=" <const_exp>
(28)         <const_type>  ::=  <integer_type>
                           |    <char_type>
                           |    <wide_char_type>
                           |    <boolean_type>
                           |    <floating_pt_type>
                           |    <string_type>
                           |    <wide_string_type>
                           |    <fixed_pt_const_type>
                           |    <scoped_name>
                           |    <octet_type>
```

```
(29)          <const_exp>  ::=  <or_expr>
(30)             <or_expr>  ::=  <xor_expr>
                            |    <or_expr> “|” <xor_expr>
(31)            <xor_expr>  ::=  <and_expr>
                            |    <xor_expr> “^” <and_expr>
(32)            <and_expr>  ::=  <shift_expr>
                            |    <and_expr> “&” <shift_expr>
(33)          <shift_expr>  ::=  <add_expr>
                            |    <shift_expr> “>>” <add_expr>
                            |    <shift_expr> “<<” <add_expr>
(34)            <add_expr>  ::=  <mult_expr>
                            |    <add_expr> “+” <mult_expr>
                            |    <add_expr> “-” <mult_expr>
(35)           <mult_expr>  ::=  <unary_expr>
                            |    <mult_expr> “*” <unary_expr>
                            |    <mult_expr> “/” <unary_expr>
                            |    <mult_expr> “%” <unary_expr>
(36)          <unary_expr>  ::=  <unary_operator> <primary_expr>
                            |    <primary_expr>
(37)    <unary_operator>  ::=  “-”
                            |    “+”
                            |    “~”
(38)        <primary_expr>  ::=  <scoped_name>
                            |    <literal>
                            |    “(” <const_exp> “)”
(39)             <literal>  ::=  <integer_literal>
                            |    <string_literal>
                            |    <wide_string_literal>
                            |    <character_literal>
                            |    <wide_character_literal>
                            |    <fixed_pt_literal>
                            |    <floating_pt_literal>
                            |    <boolean_literal>
(40)     <boolean_literal>  ::=  “TRUE”
                            |    “FALSE”
(41)<positive_int_const>  ::=  <const_exp>
```

6.9.2 Semantics

The **<scoped_name>** in the **<const_type>** production must be a previously defined name of an **<integer_type>**, **<char_type>**, **<wide_char_type>**, **<boolean_type>**, **<floating_pt_type>**, **<string_type>, <wide_string_type>**, **<octet_type>**, or **<enum_type>** constant.

Integer literals have positive integer values. Constant integer literals are considered **unsigned long** unless the value is too large, then they are considered **unsigned long long**. Unary minus is considered an operator, not a part of an integer literal. Only integer values can be assigned to integer type (**short**, **long**, **long long**) constants, and **octet** constants. Only positive integer values can be assigned to unsigned integer type constants. If the value of the right hand side of an integer constant declaration is too large to fit in the actual type of the constant on the left hand side, for example

```
const short s = 655592;
```

or is inappropriate for the actual type of the left hand side, for example

```
const octet o = -54;
```

it shall be flagged as a compile time error.

Floating point literals have floating point values. Only floating point values can be assigned to floating point type (**float**, **double**, **long double**) constants. Constant floating point literals are considered **double** unless the value is too large, then they are considered **long double**. If the value of the right hand side is too large to fit in the actual type of the constant to which it is being assigned, it shall be flagged as a compile time error. Truncation on the right for floating point types is OK.

Fixed point literals have fixed point values. Only fixed point values can be assigned to fixed point type constants. If the fixed point value in the expression on the right hand side is too large to fit in the actual fixed point type of the constant on the left hand side, then it shall be flagged as a compile time error. Truncation on the right for fixed point types is OK.

An infix operator can combine two integer types, floating point types or fixed point types, but not mixtures of these. Infix operators are applicable only to integer, floating point, and fixed point types.

Integer expressions are evaluated using the imputed type of each argument of a binary operator in turn. If either argument is **unsigned long long**, use **unsigned long long**. If either argument is **long long**, use **long long**. If either argument is **unsigned long**, use **unsigned long** (otherwise use **long**). The final result of an integer arithmetic expression must fit in the range of the declared type of the constant, otherwise an error shall be flagged by the compiler. In addition to the integer types, the final result of an integer arithmetic expression can be assigned to an **octet** constant, subject to it fitting in the range for **octet** type.

Floating point expressions are evaluated using the imputed type of each argument of a binary operator in turn. If either argument is **long double**, use **long double** (otherwise use **double**). The final result of a floating point arithmetic expression must fit in the range of the declared type of the constant, otherwise an error shall be flagged by the compiler.

Fixed-point decimal constant expressions are evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits. For example, **0123.450d** is considered to be **fixed<7,3>** and **3000.00d** is **fixed<6,2>**. Prefix operators do not affect the precision; a prefix **+** is optional, and does not change the result. The upper bounds on the number of digits and scale of the result of an infix expression, **fixed<d1,s1> op fixed<d2,s2>**, are shown in the following table.

Op	**Result: fixed<d,s>**
+	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
-	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
*	fixed<d1+d2, s1+s2>
/	fixed<(d1-s1+s2) + sinf, sinf>

A quotient may have an arbitrary number of decimal places, denoted by a scale of s_{inf}. The computation proceeds pairwise, with the usual rules for left-to-right association, operator precedence, and parentheses. All intermediate computations shall be performed using double precision (i.e., 62 digit) arithmetic. If an individual computation between a pair of fixed-point literals actually generates more than 31 significant digits, then a 31-digit result is retained as follows:

fixed<d,s> => fixed<31, 31-d+s>

Leading and trailing zeros are not considered significant. The omitted digits are discarded; rounding is not performed. The result of the individual computation then proceeds as one literal operand of the next pair of fixed-point literals to be computed.

Unary (**+ -**) and binary (*** / + -**) operators are applicable in floating-point and fixed-point expressions. Unary (**+ - ~**) and binary (*** / % + - << >> & | ^**) operators are applicable in integer expressions.

The "~" unary operator indicates that the bit-complement of the expression to which it is applied should be generated. For the purposes of such expressions, the values are 2s complement numbers. As such, the complement can be generated as follows:

Integer Constant Expression Type	Generated 2s Complement Numbers
long	long -(value+1)
unsigned long	unsigned long (2**32-1) - value
long long	long long -(value+1)
unsigned long long	unsigned long (2**64-1) - value

The "%" binary operator yields the remainder from the division of the first expression by the second. If the second operand of "%" is 0, the result is undefined; otherwise

(a/b)*b + a%b

is equal to a. If both operands are non-negative, then the remainder is non-negative; if not, the sign of the remainder is implementation dependent.

The "<<"binary operator indicates that the value of the left operand should be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 64.

The ">>" binary operator indicates that the value of the left operand should be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 64.

The "&" binary operator indicates that the logical, bitwise AND of the left and right operands should be generated.

The "|" binary operator indicates that the logical, bitwise OR of the left and right operands should be generated.

The "^" binary operator indicates that the logical, bitwise EXCLUSIVE-OR of the left and right operands should be generated.

<positive_int_const> must evaluate to a positive integer constant.

An enum constant can only be defined using a scoped name for the enumerator. The scoped name is resolved using the normal scope resolution rules Section 6.16, "Names and Scoping," on page 87. For example:

```
enum Color { red, green, blue };
const Color FAVORITE_COLOR = red;

module M {
    enum Size { small, medium, large };
};
const M::Size MYSIZE = M::medium;
```

The constant name for the RHS of an enumerated constant definition must denote one of the enumerators defined for the enumerated type of the constant. For example:

```
const Color col = red;    // is OK but
const Color another = M::medium; // is an error
```

6.10 Type Declaration

OMG IDL provides constructs for naming data types; that is, it provides C language-like declarations that associate an identifier with a type. OMG IDL uses the **typedef** keyword to associate a name with a data type; a name is also associated with a data type via the **struct**, **union**, **enum**, and **native** declarations; the syntax is:

```
(42)             <type_dcl>  ::=  “typedef” <type_declarator>
                              |   <struct_type>
                              |   <union_type>
                              |   <enum_type>
                              |   “native” <simple_declarator>
                              |   <constr_forward_decl>
(43)      <type_declarator>  ::=  <type_spec> <declarators>
```

For type declarations, OMG IDL defines a set of type specifiers to represent typed values. The syntax is as follows:

```
(44)            <type_spec>  ::=  <simple_type_spec>
                              |   <constr_type_spec>
(45) <simple_type_spec>      ::=  <base_type_spec>
                              |   <template_type_spec>
                              |   <scoped_name>
(46)      <base_type_spec>   ::=  <floating_pt_type>
                              |   <integer_type>
                              |   <char_type>
                              |   <wide_char_type>
                              |   <boolean_type>
                              |   <octet_type>
                              |   <any_type>
                              |   <object_type>
                              |   <value_base_type>
(47)<template_type_spec>::=  <sequence_type>
                              |   <string_type>
                              |   <wide_string_type>
                              |   <fixed_pt_type>
(48) <constr_type_spec>      ::=  <struct_type>
                              |   <union_type>
                              |   <enum_type>
(49)          <declarators>  ::=  <declarator> { “,” <declarator> }*
(50)           <declarator>  ::=  <simple_declarator>
                              |   <complex_declarator>
(51) <simple_declarator>     ::=  <identifier>
(52)<complex_declarator> ::=  <array_declarator>
```

The **<scoped_name>** in **<simple_type_spec>** must be a previously defined type introduced by an interface declaration (**<interface_dcl>** - see 6.7, ’Interface Declaration’), a value declaration (**<value_dcl>**, **<value_box_dcl>** or **<abstract_value_dcl>** - see 6.8, ’Value Declaration’) or a type declaration (**<type_dcl>** - see 6.10, ’Type Declaration’). Note that exceptions are not considered types in this context.

As seen above, OMG IDL type specifiers consist of scalar data types and type constructors. OMG IDL type specifiers can be used in operation declarations to assign data types to operation parameters. The next sections describe basic and constructed type specifiers.

6.10.1 Basic Types

The syntax for the supported basic types is as follows:

```
(53)   <floating_pt_type>   ::=  “float”
                             |    “double”
                             |    “long” “double”
(54)        <integer_type>   ::=  <signed_int>
                             |    <unsigned_int>
(55)          <signed_int>   ::=  <signed_short_int>
                             |    <signed_long_int>
                             |    <signed_longlong_int>
(56)  <signed_short_int>   ::=  “short”
(57)   <signed_long_int>   ::=  “long”
(58)<signed_longlong_int>::=  “long” “long”
(59)       <unsigned_int>   ::=  <unsigned_short_int>
                             |    <unsigned_long_int>
                             |    <unsigned_longlong_int>
(60)<unsigned_short_int> ::=  “unsigned” “short”
(61)<unsigned_long_int>  ::=  “unsigned” “long”
(62)<unsigned_longlong_int>::=“unsigned” “long” “long”
(63)           <char_type>   ::=  “char”
(64)    <wide_char_type>   ::=  “wchar”
(65)       <boolean_type>   ::=  “boolean”
(66)          <octet_type>   ::=  “octet”
(67)            <any_type>   ::=  “any”
```

Each OMG IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between OMG IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation. The invocation mechanism (client stub, dynamic invocation engine, and skeletons) may signal an exception condition to the client if an attempt is made to convert an illegal value. The standard system exceptions that are to be raised in such situations are defined in Section 5.6, “Exceptions,” on page 23.

6.10.1.1 Integer Types

OMG IDL integer types are **short**, **unsigned short**, **long**, **unsigned long**, **long long**, and **unsigned long long** representing integer values in the range indicated below in Table 6.12.

Table 6.12- Range of integer types

short	-2^{15} .. 2^{15} - 1
long	-2^{31} .. 2^{31} - 1
long long	-2^{63} .. 2^{63} - 1
unsigned short	0 .. 2^{16} - 1
unsigned long	0 .. 2^{32} - 1
unsigned long long	0 .. 2^{64} - 1

6.10.1.2 Floating-Point Types

OMG IDL floating-point types are **float**, **double**, and **long double**. The **float** type represents IEEE single-precision floating point numbers; the **double** type represents IEEE double-precision floating point numbers.The **long double** data type represents an IEEE double-extended floating-point number, which has an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. See *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Standard 754-1985, for a detailed specification.

6.10.1.3 Char Type

OMG IDL defines a **char** data type that is an 8-bit quantity that (1) encodes a single-byte character from any byte-oriented code set, or (2) when used in an array, encodes a multi-byte character from a multi-byte code set. In other words, an implementation is free to use any code set internally for encoding character data, though conversion to another form may be required for transmission.

The ISO 8859-1 (Latin1) character set standard defines the meaning and representation of all possible graphic characters used in OMG IDL (i.e., the space, alphabetic, digit, and graphic characters defined in Table 6.3, Table 6.4, and Table 6.5). The meaning and representation of the null and formatting characters (see Table 6.6) is the numerical value of the character as defined in the ASCII (ISO 646) standard. The meaning of all other characters is implementation-dependent.

During transmission, characters may be converted to other appropriate forms as required by a particular language binding. Such conversions may change the representation of a character but maintain the character’s meaning. For example, a character may be converted to and from the appropriate representation in international character sets.

6.10.1.4 Wide Char Type

OMG IDL defines a **wchar** data type that encodes wide characters from any character set. As with character data, an implementation is free to use any code set internally for encoding wide characters, though, again, conversion to another form may be required for transmission. The size of **wchar** is implementation-dependent.

6.10.1.5 Boolean Type

The **boolean** data type is used to denote a data item that can only take one of the values **TRUE** and **FALSE**.

6.10.1.6 Octet Type

The **octet** type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.

6.10.1.7 Any Type

The **any** type permits the specification of values that can express any OMG IDL type.

An **any** logically contains a **TypeCode** (see Section 5.5, “TypeCodes,” on page 17) and a value that is described by the **TypeCode**. Each IDL language mapping provides operations that allow programers to insert and access the **TypeCode** and value contained in an any.

6.10.2 Constructed Types

Structs, **unions** and **enums** are the constructed types. Their syntax is presented in this section:

```
(42)              <type_dcl>   ::=  “typedef” <type_declarator>
                               |    <struct_type>
                               |    <union_type>
                               |    <enum_type>
                               |    “native” <simple_declarator>
                               |    <constr_forward_decl>
(48) <constr_type_spec>   ::=  <struct_type>
                               |    <union_type>
                               |    <enum_type>
(99)<constr_forward_decl>::=  “struct” <identifier>
                               |    “union” <identifier>
```

6.10.2.1 Structures

The syntax for **struct** type is

```
(69)           <struct_type>   ::=  “struct” <identifier> “{” <member_list> “}”
(70)           <member_list>   ::=  <member>+
(71)                <member>   ::=  <type_spec> <declarators> “;”
```

The **<identifier>** in **<struct_type>** defines a new legal type. Structure types may also be named using a **typedef** declaration.

Name scoping rules require that the member declarators in a particular structure be unique. The value of a **struct** is the value of all of its members.

6.10.2.2 Discriminated Unions

The discriminated **union** syntax is:

```
(72)            <union_type>   ::=  “union” <identifier> “switch”
                               “(” <switch_type_spec> “)”
                               “{” <switch_body> “}”
```

```
(73) <switch_type_spec> ::= <integer_type>
                        |   <char_type>
                        |   <boolean_type>
                        |   <enum_type>
                        |   <scoped_name>
(74)      <switch_body> ::= <case>+
(75)             <case> ::= <case_label>+ <element_spec> ";"
(76)       <case_label> ::= "case" <const_exp> ":"
                        |   "default" ":"
(77)     <element_spec> ::= <type_spec> <declarator>
```

OMG IDL unions are a cross between the C `union` and `switch` statements. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. The **<identifier>** following the **union** keyword defines a new legal type. Union types may also be named using a **typedef** declaration. The **<const_exp>** in a **<case_label>** must be consistent with the **<switch_type_spec>**. A **default** case can appear at most once. The **<scoped_name>** in the **<switch_type_spec>** production must be a previously defined **integer**, **char**, **boolean**, or **enum** type.

Case labels must match or be automatically castable to the defined type of the discriminator. Name scoping rules require that the element declarators in a particular union be unique. If the **<switch_type_spec>** is an **<enum_type>**, the identifier for the enumeration is in the scope of the union; as a result, it must be distinct from the element declarators.

It is not required that all possible values of the union discriminator be listed in the **<switch_body>**. The value of a union is the value of the discriminator together with one of the following:

- If the discriminator value was explicitly listed in a **case** statement, the value of the element associated with that **case** statement.
- If a default **case** label was specified, the value of the element associated with the default **case** label.
- No additional value.

The values of the constant expressions for the case labels of a single union definition must be distinct. A union type can contain a default label only where the values given in the non-default labels do not cover the entire range of the union's discriminant type.

Access to the discriminator and the related element is language-mapping dependent.

While any ISO Latin-1 (8859.1) IDL character literal may be used in a **<case_label>** in a union definition whose discriminator type is **char**, not all of these characters are present in all transmission code sets that may be negotiated by GIOP or in all native code sets that may be used by implementation language compilers and runtimes. When an attempt is made to marshal to CDR a **union** whose discriminator value of **char** type is not available in the negotiated transmission code set, or to demarshal from CDR a **union** whose discriminator value of **char** type is not available in the native code set, a **DATA_CONVERSION** system exception is raised. Therefore, to ensure portability and interoperability, care must be exercised when assigning the **<case_label>** for a **union** member whose discriminator type is **char**. Due to these issues, use of **char** types as the discriminator type for **union**s is not recommended.

6.10.2.3 Constructed Recursive Types and IForward Declarations

The IDL syntax allows the generation of recursive structures and unions via members that have a sequence type. The element type of a recursive sequence struct or union member must identify a struct, union, or valuetype. (A valuetype is allowed to have a member of its own type either directly or indirectly through a member of a constructed type—see Section 6.8.1.6, “Value Type Example,” on page 61.) For example, the following is legal:

```
struct Foo {
    long value;
    sequence<Foo> chain;      // Deprecated (see  Section 6.10.6)
}
```

See Section 6.10.3.1, “Sequences,” on page 75 for details of the **sequence** template type.

IDL supports recursive types via a forward declaration for structures and unions (as well as for valuetypes—see Section 6.8.1.6, “Value Type Example,” on page 61). Because anonymous types are deprecated (see Section 6.10.6, “Deprecated Anonymous Types,” on page 78), the previous example is better written as:

```
struct Foo;                              // Forward declaration
typedef sequence<Foo> FooSeq;
struct Foo {
    long value;
    FooSeq chain;
};
```

The forward declaration for the structure enables the definition of the sequence type **FooSeq**, which is used as the type of the recursive member.

Forward declarations are legal for structures and unions.A structure or union type is termed incomplete until its full definition is provided; that is, until the scope of the structure or union definition is closed by a terminating "}". For example:

```
struct Foo;        // Introduces Foo type name,
                   // Foo is incomplete now
                   // ...
struct Foo {
    // ...
};                 // Foo is complete at this point
```

If a structure or union is forward declared, a definition of that structure or union must follow the forward declaration in the same source file. Compilers shall issue a diagnostic if this rule is violated. Multiple forward declarations of the same structure or union are legal.

If a recursive structure or union member is used, sequence members that are recursive must refer to an incomplete type currently under definition. For example:

```
struct Foo;                // Forward declaration
typedef sequence<Foo> FooSeq;
struct Bar {
    long value;
    FooSeq chain;          //Illegal, Foo is not an enclosing struct or union
};
```

Compilers shall issue a diagnostic if this rule is violated.

Recursive definitions can span multiple levels. For example:

```
union Bar;                      // Forward declaration
        typedef sequence<Bar> BarSeq;
union Bar switch(long) {    // Define incomplete union
    case 0:
        long l_mem;
    case 1:
        struct Foo {
            double d_mem;
            BarSeq nested;      // OK, recurse on enclosing
                                // incomplete type
    } s_mem;
};
```

An incomplete type can only appear as the element type of a sequence definition. A sequence with incomplete element type is termed an *incomplete sequence type*:

```
struct Foo;                             // Forward declaration
typedef sequence<Foo> FooSeq;       // incomplete
```

An incomplete sequence type can appear only as the element type of another sequence, or as the member type of a structure or union definition. For example:

```
struct Foo;                             // Forward declaration
typedef sequence<Foo> FooSeq;       // OK
typedef sequence<FooSeq> FooTree; // OK

interface I {
    FooSeq op1();       // Illegal, FooSeq is incomplete
    void op2(           // Illegal, FooTree is incomplete
        in FooTree t
    );
};

struct Foo {            // Provide definition of Foo
    long l_mem;
    FooSeq chain;       // OK
    FooTree tree;       // OK
};

interface J {
    FooSeq op1();       // OK, FooSeq is complete
    void op2(
        in FooTree t    // OK, FooTree is complete
    );
};
```

Compilers shall issue a diagnostic if this rule is violated.

6.10.2.4 Enumerations

Enumerated types consist of ordered lists of identifiers. The syntax is:

```
(78)          <enum_type>   ::=  "enum" <identifier>
                                 "{" <enumerator> { "," <enumerator> }* "}"
(79)         <enumerator>   ::=  <identifier>
```

A maximum of 2^{32} identifiers may be specified in an enumeration; as such, the enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers. Any language mapping that permits two enumerators to be compared or defines successor/predecessor functions on enumerators must conform to this ordering relation. The **<identifier>** following the **enum** keyword defines a new legal type. Enumerated types may also be named using a **typedef** declaration.

6.10.3 Template Types

The template types are:

```
(47)<template_type_spec>::=  <sequence_type>
                        |    <string_type>
                        |    <wide_string_type>
                        |    <fixed_pt_type>
```

6.10.3.1 Sequences

OMG IDL defines the sequence type **sequence**. A sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time).

The syntax is:

```
(80)     <sequence_type>   ::=  "sequence" "<" <simple_type_spec> "," <positive_int_const> ">"
                           |    "sequence" "<" <simple_type_spec> ">"
```

The second parameter in a sequence declaration indicates the maximum size of the sequence. If a positive integer constant is specified for the maximum size, the sequence is termed a bounded sequence. If no maximum size is specified, size of the sequence is unspecified (unbounded).

Prior to passing a bounded or unbounded sequence as a function argument (or as a field in a structure or union), the length of the sequence must be set in a language-mapping dependent manner. After receiving a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

A sequence type may be used as the type parameter for another sequence type. For example, the following:

```
typedef sequence< sequence<long> > Fred;
```

declares Fred to be of type "unbounded sequence of unbounded sequence of long." Note that for nested sequence declarations, white space must be used to separate the two ">" tokens ending the declaration so they are not parsed as a single ">>" token.

6.10.3.2 Strings

OMG IDL defines the string type **string** consisting of all possible 8-bit quantities except null. A string is similar to a sequence of char. As with sequences of any type, prior to passing a string as a function argument (or as a field in a structure or union), the length of the string must be set in a language-mapping dependent manner. The syntax is:

```
(81)        <string_type>   ::=  “string” “<” <positive_int_const> “>”
                            |    “string”
```

The argument to the string declaration is the maximum size of the string. If a positive integer maximum size is specified, the string is termed a bounded string; if no maximum size is specified, the string is termed an unbounded string.

Strings are singled out as a separate type because many languages have special built-in functions or standard library functions for string manipulation. A separate string type may permit substantial optimization in the handling of strings compared to what can be done with sequences of general types.

6.10.3.3 Wstrings

The **wstring** data type represents a sequence of wchar, except the wide character null. The type wstring is similar to that of type string, except that its element type is wchar instead of char. The actual length of a wstring is set at run-time and, if the bounded form is used, must be less than or equal to the bound.

The syntax for defining a wstring is:

```
(82)  <wide_string_type>   ::=  “wstring” “<” <positive_int_const> “>”
                           |    “wstring”
```

6.10.3.4 Fixed Type

The **fixed** data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is a non-negative integer less than or equal to the total number of digits (note that constants with effectively negative scale, such as 10000, are always permitted).

The **fixed** data type will be mapped to the native fixed point capability of a programming language, if available. If there is not a native fixed point type, then the IDL mapping for that language will provide a fixed point data type. Applications that use the IDL fixed point type across multiple programming languages must take into account differences between the languages in handling rounding, overflow, and arithmetic precision.

The syntax of fixed type is:

```
(96)       <fixed_pt_type>       ::=“fixed” “<“ <positive_int_const> “,” <positive_int_const> “>”
(97) <fixed_pt_const_type>   ::=“fixed”
```

6.10.4 Complex Declarator

6.10.4.1 Arrays

OMG IDL defines multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension.

The syntax for arrays is:

(83) <array_declarator> ::= <identifier> <fixed_array_size>$^+$

(84) <fixed_array_size> ::= "[" <positive_int_const> "]"

The array size (in each dimension) is fixed at compile time. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

The implementation of array indices is language mapping specific; passing an array index as a parameter may yield incorrect results.

6.10.5 Native Types

OMG IDL provides a declaration for use by object adapters to define an opaque type whose representation is specified by the language mapping for that object adapter.

The syntax is:

(42) <type_dcl> ::="native" <simple_declarator>

(51) <simple_declarator>::=<identifier>

This declaration defines a new type with the specified name. A native type is similar to an IDL basic type. The possible values of a native type are language-mapping dependent, as are the means for constructing them and manipulating them. Any interface that defines a native type requires each language mapping to define how the native type is mapped into that programming language.

A native type may be used only to define operation parameters, results, and exceptions. If a native type is used for an exception, it must be mapped to a type in a programming language that can be used as an exception. Native type parameters are permitted only in operations of **local interface**s or **valuetype**s. Any attempt to transmit a value of a native type in a remote invocation may raise the MARSHAL standard system exception.

It is recommended that native types be mapped to equivalent type names in each programming language, subject to the normal mapping rules for type names in that language. For example, in a hypothetical Object Adapter IDL module:

```
module HypotheticalObjectAdapter {
    native Servant;
    interface HOA {
        Object activate_object(in Servant x);
    };
};
```

The IDL type Servant would map to **HypotheticalObjectAdapter::Servant** in C++ and the **activate_object** operation would map to the following C++ member function signature:

```
    CORBA::Object_ptr activate_object(
    HypotheticalObjectAdapter::Servant x);
```

The definition of the C++ type `HypotheticalObjectAdapter::Servant` would be provided as part of the C++ mapping for the HypotheticalObjectAdapter module.

Note – The native type declaration is provided specifically for use in object adapter interfaces, which require parameters whose values are concrete representations of object implementation instances. It is strongly recommended that it not be used in service or application interfaces. The native type declaration allows object adapters to define new primitive types without requiring changes to the OMG IDL language or to OMG IDL compiler.

6.10.6 Deprecated Anonymous Types

IDL currently permits the use of anonymous types in a number of places. For example:

```
struct Foo {
    long value;
    sequence<Foo> chain;     // Legal (but deprecated)
}
```

Anonymous types cause a number of problems for language mappings and are therefore deprecated by this specification. Anonymous types will be removed in a future version, so new IDL should avoid use of anonymous types and use a typedef to name such types instead. Compilers need not issue a warning if a deprecated construct is encountered.

The following (non-exhaustive) examples illustrate deprecated uses of anonymous types.

Anonymous bounded string and bounded wide string types are deprecated. This rule affects constant definitions, attribute declarations, return value and parameter type declarations, sequence and array element declarations, and structure, union, exception, and valuetype member declarations. For example:

```
const string<5> GREETING = “Hello”;          // Deprecated

interface Foo {
    readonly attribute wstring<5> name;      // Deprecated
    wstring<5> op(in wstring<5> param);      // Deprecated
};
typedef sequence<wstring<5> > WS5Seq;        // Deprecated
typedef wstring<5> NameVector [10];          // Deprecated
struct A {
    wstring<5> mem;                          // Deprecated
};
// Anonymous member type in unions, exceptions,
// and valuetypes are deprecated as well.
```

This is better written as:

```
typedef string<5> GreetingType;
const GreetingType GREETING = “Hello”;

typedef wstring<5> ShortWName;
interface Foo {
    readonly attribute ShortWName name;
    ShortWName op(in ShortWName param);
};
typedef sequence<ShortWName> NameSeq;
typedef ShortWName NameVector[10];
```

```
struct A {
    GreetingType mem;
};
```

Anonymous fixed-point types are deprecated. This rule affects attribute declarations, return value and parameter type declarations, sequence and array element declarations, and structure, union, exception, and valuetype member declarations.

```
struct Foo {
    fixed<10,5> member;                    // Deprecated
};
```

This is better written as:

```
typedef fixed<10,5> MyType;
struct Foo {
    MyType member;
};
```

Anonymous member types in structures, unions, exceptions, and valuetypes are deprecated:

```
union U switch(long) {
    case 1:
        long array_mem[10];                // Deprecated
    case 2:
        sequence<long> seq_mem;            // Deprecated
    case 3:
        string<5> bstring_mem;
};
```

This is better written as:

```
typedef long LongArray[10];
typedef sequence<long> LongSeq;
typedef string<5> ShortName;
union U switch (long) {
    case 1:
        LongArray array_mem;
    case 2:
        LongSeq seq_mem;
    case 3:
        ShortName bstring_mem;
                        };
```

Anonymous array and sequence elements are deprecated:

```
typedef sequence<sequence<long> > NumberTree;   // Deprecated
typedef fixed<10,2> FixedArray[10];
```

This is better written as:

```
typedef sequence<long> ListOfNumbers;
typedef sequence<ListOfNumbers> NumberTree;
typedef fixed<10,2> Fixed_10_2;
typedef Fixed_10_2 FixedArray[10];
```

The preceding examples are not exhaustive. They simply illustrate the rule that, for a type to be used in the definition of another type, constant, attribute, return value, parameter, or member, that type must have a name. Note that the following example is not deprecated (even though stylistically poor):

```
struct Foo {
    struct Bar {
        long l_mem;
        double d_mem;
    } bar_mem_1;          // OK, not anonymous
    Bar bar_mem_2;        // OK, not anonymous
};
typedef sequence<Foo::Bar> FooBarSeq;     // Scoped names are OK
```

6.11 Exception Declaration

Exception declarations permit the declaration of struct-like data structures, which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is as follows:

```
(86)          <except_dcl>   ::=  "exception" <identifier> "{" <member>* "}"
```

Each exception is characterized by its OMG IDL identifier, an exception type identifier, and the type of the associated return value (as specified by the **<member>** in its declaration). If an exception is returned as the outcome to a request, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

An identifier declared to be an exception identifier may thereafter appear only in a raises clause of an operation declaration, and nowhere else.

A set of standard system exceptions is defined corresponding to standard run-time errors, which may occur during the execution of a request. These standard system exceptions are documented in Section 5.6, “Exceptions,” on page 23.

6.12 Operation Declaration

Operation declarations in OMG IDL are similar to C function declarations. The syntax is:

```
(87)              <op_dcl>   ::=  [ <op_attribute> ] <op_type_spec>
                                  <identifier> <parameter_dcls>
                                  [ <raises_expr> ] [ <context_expr> ]
(88)        <op_attribute>   ::=  "oneway"
(89)        <op_type_spec>   ::=  <param_type_spec>
                             |    "void"
```

An operation declaration consists of:

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in Section 6.12.1, “Operation Attribute,” on page 81.
- The type of the operation’s return result; the type may be any type that can be defined in OMG IDL. Operations that do not return a result must specify the **void** type.
- An identifier that names the operation in the scope of the interface in which it is defined.
- A parameter list that specifies zero or more parameter declarations for the operation. Parameter declaration is described in Section 6.12.2, “Parameter Declarations,” on page 81.
- An optional raises expression that indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in Section 6.12.3, “Raises Expressions,” on page 82.
- An optional context expression that indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in Section 6.12.4, “Context Expressions,” on page 83.

Some implementations and/or language mappings may require operation-specific pragmas to immediately precede the affected operation declaration.

6.12.1 Operation Attribute

The operation attribute specifies which invocation semantics the communication service must provide for invocations of a particular operation. An operation attribute is optional. The syntax for its specification is as follows:

```
(88)        <op_attribute>   ::=  “oneway”
```

When a client invokes an operation with the **oneway** attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once. An operation with the **oneway** attribute must not contain any output parameters and must specify a **void** return type. An operation defined with the **oneway** attribute may not include a raises expression; invocation of such an operation, however, may raise a standard system exception.

If an **<op_attribute>** is not specified, the invocation semantics is at-most-once if an exception is raised; the semantics are exactly-once if the operation invocation returns successfully.

6.12.2 Parameter Declarations

Parameter declarations in OMG IDL operation declarations have the following syntax:

```
(90)    <parameter_dcls>   ::=  “(” <param_dcl> { “,” <param_dcl> }* “)”
                           |    “(” “)”
(91)         <param_dcl>   ::=  <param_attribute> <param_type_spec> <simple_declarator>
(92)   <param_attribute>   ::=  “in”
                           |    “out”
                           |    “inout”
(95) <param_type_spec>     ::=  <base_type_spec>
                           |    <string_type>
                           |    <wide_string_type>
                           |    <scoped_name>
```

A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. The directional attributes are:

- **in** - the parameter is passed from client to server.
- **out** - the parameter is passed from server to client.
- **inout** - the parameter is passed in both directions.

It is expected that an implementation will *not* attempt to modify an **in** parameter. The ability to even attempt to do so is language-mapping specific; the effect of such an action is undefined.

If an exception is raised as a result of an invocation, the values of the return result and any **out** and **inout** parameters are undefined.

6.12.3 Raises Expressions

There are two kinds of raises expressions as described in this section.

6.12.3.1 Raises Expression

A **raises** expression specifies which exceptions may be raised as a result of an invocation of the operation or accessing (invoking the _get operation of) a readonly attribute. The syntax for its specification is as follows:

```
(93)        <raises_expr>   ::=  "raises" "(" <scoped_name>
                                 { ","  <scoped_name> }* ")"
```

The **<scoped_name>**s in the **raises** expression must be previously defined exceptions or native types. If a native type is used as an exception for an operation, the operation must appear in either a local interface or a valuetype.

In addition to any operation-specific exceptions specified in the **raises** expression, there are a standard set of system exceptions that may be signalled by the ORB. These standard system exceptions are described in Section 5.6.3, "Standard System Exception Definitions," on page 25. However, standard system exceptions may *not* be listed in a **raises** expression.

The absence of a **raises** expression on an operation implies that there are no operation-specific exceptions. Invocations of such an operation are still liable to receive one of the standard system exceptions.

6.12.3.2 getraises and setraises Expressions

getraises and **setraises** expressions specify which exceptions may be raised as a result of an invocation of the accessor (**_get**) and a mutator (**_set**) functions of an attribute. The syntax for its specification is as follows:

```
(108)  <attr_raises_expr>   ::=  <get_excep_expr> [ <set_excep_expr> ]
                            |    <set_excep_expr>
(109)  <get_excep_expr>     ::=  "getraises" <exception_list>
(110)  <set_excep_expr>     ::=  "setraises" <exception_list>
(111)      <exception_list  ::=  "(" <scoped_name>
                                  { "," <scoped_name> } * ")"
```

The **<scoped_name>**s in the **getraises** and **setraises** expressions must be previously defined exceptions.

In addition to any attribute-specific exceptions specified in the **getraises** and **setraises** expressions, there are a standard set of exceptions that may be signalled by the ORB. These standard exceptions are described in Section 5.6.3, “Standard System Exception Definitions,” on page 25. However, standard exceptions may *not* be listed in a **getraises** or **setraises** expression.

The absence of a **getraises** or **setraises** expression on an attribute implies that there are no accessor-specific or mutator-exceptions respectively. Invocations of such an accessor or mutator are still liable to receive one of the standard exceptions.

Note – The exceptions associated with the accessor operation corresponding to a **readonly attribute** is specified using a simple **raises** expression as specified in Section 6.12.3.1, “Raises Expression,” on page 82. The **getraises** and **setraises** expressions are used only in **attribute**s that are not **readonly**.

6.12.4 Context Expressions

A **context** expression specifies which elements of the client’s context may affect the performance of a request by the object. The syntax for its specification is as follows:

```
(94)      <context_expr>   ::=  “context” “(” <string_literal>
                                { “,” <string_literal> }* “)”
```

The run-time system guarantees to make the value (if any) associated with each **<string_literal>** in the client’s context available to the object implementation when the request is delivered. The ORB and/or object is free to use information in this *request context* during request resolution and performance.

The absence of a context expression indicates that there is no request context associated with requests for this operation.

Each **string_literal** is a non-empty string. If the character ‘*’ appears in a **string_literal**, it must appear only once, as the last character of **string_literal**, and must be preceded by one or more characters other than ‘*’.

The mechanism by which a client associates values with the context identifiers.

6.13 Attribute Declaration

An interface can have attributes as well as operations; as such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions; one to retrieve the value of the attribute and one to set the value of the attribute.

The syntax for **attribute** declaration is:

```
(85)            <attr_dcl>   ::=  <readonly_attr_spec>
                             |    <attr_spec>
(104)<readonly_attr_spec> ::=  “readonly” “attribute” <param_type_spec>  <readonly_attr_declarator>
(105)<readonly_attr_declarator>::=  <simple_declarator> <raises_expr>
                             |    <simple_declarator>
                                  { “,” <simple_declarator> }*
(106)           <attr_spec>   ::= “attribute” <param_type_spec> <attr_declarator>
(107)    <attr_declarator>   ::=  <simple_declarator> <attr_raises_expr>
                             |    <simple_declarator>
                                   { “,” <simple_declarator> }*
```

The optional **readonly** keyword indicates that there is only a single accessor function—the retrieve value function. Consider the following example:

```
interface foo {
    enum material_t {rubber, glass};
    struct position_t {
        float x, y;
    };

    attribute float radius;
    attribute material_t material;
    readonly attribute position_t position;
    • • •
};
```

The attribute declarations are equivalent to the following pseudo-specification fragment, assuming that one of the leading '_'s is removed by application of the Escaped Identifier rule described in Section 6.2.3.1, "Escaped Identifiers," on page 44:

```
• • •
float       __get_radius ();
void        __set_radius (in float r);
material_t  __get_material ();
void        __set_material (in material_t m);
position_t  __get_position ();
• • •
```

The actual accessor function names are language-mapping specific. The attribute name is subject to OMG IDL's name scoping rules; the accessor function names are guaranteed *not* to collide with any legal operation names specifiable in OMG IDL.

Attributes are inherited. An attribute name *cannot* be redefined to be a different type. See Section 6.15, "CORBA Module," on page 86 for more information on redefinition constraints and the handling of ambiguity.

6.14 Repository Identity Related Declarations

Two constructs that are provided for specifying information related to Repository Id are described in this section.

6.14.1 Repository Identity Declaration

The syntax of a repository identity declaration is as follows:

(102) <type_id_dcl> ::= "typeid" <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeid**
- a *<scoped_name>* that denotes the named IDL construct to which the repository identifier is assigned
- a string literal that must contain a valid repository identifier value

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface
- component
- home
- facet
- receptacle
- event sink
- event source
- finder
- factory
- event type
- value type
- value type member
- value box
- constant
- typedef
- exception
- attribute
- operation
- enum
- local

The value of the string literal is assigned as the repository identity of the specified type definition. This value will be returned as the **RepositoryId** by the interface repository definition object corresponding to the specified type definition. Language mapping constructs, such as Java helper classes, that return repository identifiers shall return the values declared for their corresponding definitions.

At most one repository identity declaration may occur for any named type definition. An attempt to redefine the repository identity for a type definition is illegal, regardless of the value of the redefinition.

If no explicit repository identity declaration exists for a type definition, the repository identifier for the type definition shall be an IDL format repository identifier, as defined in Section 4.2, “OMG IDL Format,” on page 2.

6.14.2 Repository Identifier Prefix Declaration

The syntax of a repository identifier prefix declaration is as follows:

(103) <type_prefix_dcl> ::= “typeprefix” <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeprefix**

- a <*scoped_name*> that denotes an IDL name scope to which the prefix applies
- a string literal that must contain the string to be prefixed to repository identifiers in the specified name scope

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface (including abstract or local interface)
- value type (including abstract, custom, and box value types)
- event type (including abstract and custom value types)
- specification scope (**::**)

The specified string is prefixed to the body of all repository identifiers in the specified name scope, whose values are assigned by default. The specified string shall be a list of one or more identifiers, separated by the "/" characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore ("_"), hyphen ("-"), and period (".") characters. The string shall not contain a trailing slash ("/"), and it shall not begin with the characters underscore ("_"), hyphen ("-") or period ("."). To elaborate:

By "prefixed to the body of a repository identifier," we mean that the specified string is inserted into the default IDL format repository identifier immediately after the format name and colon ("IDL:") at the beginning of the identifier. A forward slash ('/') character is inserted between the end of the specified string and the remaining body of the repository identifier.

The prefix is only applied to repository identifiers whose values are not explicitly assigned by a typeid declaration. The prefix is applied to all such repository identifiers in the specified name scope, including the identifier of the construct that constitutes the name scope.

6.14.3 Repository Id Conflict

In IDL that contains both pragma prefix/ID declarations and typeprefix/typeid declarations if the repository id for an IDL element computed by using pragmas and typeid/typeprefix are not identical, it is an error. Note that this rule applies only when the repository id value computation uses explicitly declared values from declarations of both kinds. If the repository id computed using explicitly declared values of one kind conflicts with one computed with implicit values of the other kind, the repository id based on explicitly declared values shall prevail.

6.15 CORBA Module

Names defined by the CORBA specification are in a module named CORBA. In an OMG IDL specification, however, OMG IDL keywords such as **Object** must not be preceded by a "**CORBA::**" prefix. Other interface names such as **TypeCode** are not OMG IDL keywords, so they must be referred to by their fully scoped names (e.g., **CORBA::TypeCode**) within an OMG IDL specification.

For example in:

```
#include <orb.idl>
module M {
    typedef CORBA::Object myObjRef;     // Error: keyword Object scoped
    typedef TypeCode myTypeCode;        // Error: TypeCode undefined
    typedef CORBA::TypeCode TypeCode;// OK
};
```

The file **orb.idl** contains the IDL definitions for the **CORBA** module. Except for **CORBA::TypeCode**, the file **orb.idl** must be included in IDL files that use names defined in the **CORBA** module. IDL files that use **CORBA::TypeCode** may obtain its definition by including either the file **orb.idl** or the file **TypeCode.idl**.

The exact contents of **TypeCode.idl** are implementation dependent. One possible implementation of **TypeCode.idl** may be:

```
// PIDL
#ifndef _TYPECODE_IDL_
#define _TYPECODE_IDL_
#pragma prefix "omg.org"
module CORBA {
    interface TypeCode;
};
#endif // _TYPECODE_IDL_
```

For IDL compilers that implicitly define **CORBA::TypeCode**, **TypeCode.idl** could consist entirely of a comment as shown below:

```
// PIDL
// CORBA::TypeCode implicitly built into the IDL compiler
// Hence there are no declarations in this file
```

Because the compiler implicitly contains the required declaration, this file meets the requirement for compliance.

The version of **CORBA** specified in this release of the specification is version **<x.y>**, and this is reflected in the IDL for the **CORBA** module by including the following pragma version (see Section 4.3.3, “The Version Pragma,” on page 6):

```
    #pragma version CORBA <x.y>
```

as the first line immediately following the very first **CORBA** module introduction line, which in effect associates that version number with the **CORBA** entry in the **IR**. The version number in that version pragma line must be changed whenever any changes are made to any remotely accessible parts of the **CORBA** module in an officially released OMG standard.

6.16 Names and Scoping

OMG IDL identifiers are case insensitive; that is, two identifiers that differ only in the case of their characters are considered redefinitions of one another. However, all references to a definition must use the same case as the defining occurrence. This allows natural mappings to case-sensitive languages. For example:

```
module M {
    typedef long Long;      // Error: Long clashes with keyword long
    typedef long TheThing;
    interface I {
        typedef long MyLong;
        myLong op1(                  // Error: inconsistent capitalization
            in TheThing thething;    // Error: TheThing clashes with thething
        );
    };
};
```

6.16.1 Qualified Names

A qualified name (one of the form <scoped-name>::<identifier>) is resolved by first resolving the qualifier <scoped-name> to a scope S, and then locating the definition of <identifier> within S. The identifier must be directly defined in S or (if S is an interface) inherited into S. The <identifier> is not searched for in enclosing scopes.

When a qualified name begins with "::", the resolution process starts with the file scope and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph.

Every OMG IDL definition in a file has a global name within that file. The global name for a definition is constructed as follows.

Prior to starting to scan a file containing an OMG IDL specification, the name of the current root is initially empty ("") and the name of the current scope is initially empty (""). Whenever a **module** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current root; upon detection of the termination of the **module**, the trailing "::" and identifier are deleted from the name of the current root. Whenever an **interface**, **struct**, **union**, or **exception** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current scope; upon detection of the termination of the **interface**, **struct**, **union**, or **exception**, the trailing "::" and identifier are deleted from the name of the current scope. Additionally, a new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited.

The global name of an OMG IDL definition is the concatenation of the current root, the current scope, a "::", and the <identifier>, which is the local name for that definition.

Note that the global name in an OMG IDL file corresponds to an absolute **ScopedName** in the Interface Repository.

Inheritance causes all identifiers defined in base interfaces, both direct and indirect, to be visible in derived interfaces. Such identifiers are considered to be semantically the same as the original definition. Multiple paths to the same original identifier (as results from the diamond shape in Figure 6.1) do not conflict with each other.

Inheritance introduces multiple global OMG IDL names for the inherited identifiers. Consider the following example:

```
interface A {
    exception E {
        long L;
    };
    void f() raises(E);
};

interface B: A {
    void g() raises(E);
};
```

In this example, the exception is known by the global names **::A::E** and **::B::E**.

Ambiguity can arise in specifications due to the nested naming scopes. For example:

```
interface A {
    typedef string<128> string_t;
};

interface B {
    typedef string<256> string_t;
};

interface C: A, B {
    attribute string_t        Title;          // Error: Ambiguous
    attribute A::string_t     Name;           // OK
    attribute B::string_t     City;           // OK
};
```

The declaration of attribute **Title** in interface **C** is ambiguous, since the compiler does not know which **string_t** is desired. Ambiguous declarations yield compilation errors.

6.16.2 Scoping Rules and Name Resolution

Contents of an entire OMG IDL file, together with the contents of any files referenced by #include statements, forms a naming scope. Definitions that do not appear inside a scope are part of the global scope. There is only a single global scope, irrespective of the number of source files that form a specification.

The following kinds of definitions form scopes:

- module
- interface
- valuetype
- struct
- union
- operation
- exception
- eventtype
- component
- home

The scope for module, interface, valuetype, struct, exception, eventtype, component, and home begins immediately following its opening '{' and ends immediately preceding its closing '}'. The scope of an operation begins immediately following its '(' and ends immediately preceding its closing ')'. The scope of a union begins immediately following the '(' following the keyword **switch**, and ends immediately preceding its closing '}'. The appearance of the declaration of any of these kinds in any scope, subject to semantic validity of such declaration, opens a nested scope associated with that declaration.

An identifier can only be defined once in a scope. However, identifiers can be redefined in nested scopes. An identifier declaring a module is considered to be defined by its first occurrence in a scope. Subsequent occurrences of a module declaration with the same identifier within the same scope reopens the module and hence its scope, allowing additional definitions to be added to it.

The name of an interface, value type, struct, union, exception, or a module may not be redefined within the immediate scope of the interface, value type, struct, union, exception, or the module. For example:

```
module M {
    typedef short M;          // Error: M is the name of the module
                              //          in the scope of which the typedef is.
    interface I {
        void i (in short j);  // Error: i clashes with the interface name I
    };
};
```

An identifier from a surrounding scope is introduced into a scope if it is used in that scope. An identifier is not introduced into a scope by merely being visible in that scope. The use of a scoped name introduces the identifier of the outermost scope of the scoped name. For example in:

```
module M {
    module Inner1 {
        typedef string S1;
    };

    module Inner2 {
        typedef string inner1;      // OK
    };
}
```

The declaration of **Inner2::inner1** is OK because the identifier **Inner1**, while visible in module **Inner2**, has not been introduced into module **Inner2** by actual use of it. On the other hand, if module **Inner2** were:

```
module Inner2{
    typedef Inner1::S1 S2;     // Inner1 introduced
    typedef string inner1;     // Error
    typedef string S1;         // OK
};
```

The definition of **inner1** is now an error because the identifier **Inner1** referring to the **module Inner1** has been introduced in the scope of module **Inner2** in the first line of the module declaration. Also, the declaration of **S1** in the last line is OK since the identifier **S1** was not introduced into the scope by the use of **Inner1::S1** in the first line.

Only the first identifier in a qualified name is introduced into the current scope. This is illustrated by **Inner1::S1** in the example above, which introduces "**Inner1**" into the scope of "**Inner2**" but does not introduce "**S1**." A qualified name of the form "**::X::Y::Z**" does not cause "**X**" to be introduced, but a qualified name of the form "**X::Y::Z**" does.

Enumeration value names are introduced into the enclosing scope and then are treated like any other declaration in that scope. For example:

```
interface A {
    enum E { E1, E2, E3 };     // line 1
    enum BadE { E3, E4, E5 }; // Error: E3 is already introduced
                               // into the A scope in line 1 above
};

interface C {
    enum AnotherE { E1, E2, E3 };
};

interface D : C, A {
    union U switch ( E ) {
        case A::E1 : boolean b;// OK.
        case E2 : long l;      // Error: E2 is ambiguous (notwithstanding
                               // the switch type specification!!)
    };
};
```

Type names defined in a scope are available for immediate use within that scope. In particular, see Section 6.10.2, “Constructed Types,” on page 71 on cycles in type definitions.

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes, while taking into consideration inheritance relationships among interfaces. For example:

```
module M {
    typedef long ArgType;
    typedef ArgType AType;          // line l1
    interface B {
        typedef string ArgType;     // line l3
        ArgType opb(in AType i);    // line l2
    };
};

module N {
    typedef char ArgType;           // line l4
    interface Y : M::B {
        void opy(in ArgType i);     // line l5
    };
};
```

The following scopes are searched for the declaration of **ArgType** used on **line l5**:

1. Scope of **N::Y** before the use of **ArgType**.

2. Scope of **N::Y**’s base interface **M::B**. (inherited scope)

3. Scope of **module N** before the definition of **N::Y**.

4. Global scope before the definition of **N**.

M::B::ArgType is found in **step 2** in **line l3**, and that is the definition that is used in **line l5**, hence **ArgType** in **line l5** is **string**. It should be noted that **ArgType** is not **char** in **line l5**. Now if **line l3** were removed from the definition of interface **M::B**, then **ArgType** on **line l5** would be **char** from **line l4**, which is found in **step 3**.

Following analogous search steps for the types used in the operation **M::B::opb** on **line l2**, the type of **AType** used on **line l2** is **long** from the **typedef** in **line l1** and the return type **ArgType** is **string** from **line l3**.

6.16.3 Special Scoping Rules for Type Names

Once a type has been defined anywhere within the scope of a module, interface, or valuetype it may not be redefined except within the scope of a nested module, interface, or valuetype, or within the scope of a derived interface or valuetype. For example:

```
typedef short TempType;          // Scope of TempType begins here

module M {
      typedef string ArgType;   // Scope of ArgType begins here
      struct S {
            ::M::ArgType a1;      // Nothing introduced here
            M::ArgType a2;       // M introduced here
            ::TempType temp;      // Nothing introduced here
      };                       // Scope of (introduced) M ends here
      // ...
};                              // Scope of ArgType ends here

// Scope of global TempType ends here (at end of file)
```

The scope of an introduced type name is from the point of introduction to the end of its enclosing scope.

However, if a *type* name is *introduced* into a scope that is nested in a non-module scope definition, its *potential* scope extends over all its enclosing scopes out to the enclosing non-module scope. (For types that are defined outside an inon-module scope, the scope and the potential scope are identical.) For example:

```
module M {
      typedef long ArgType;
      const long I = 10;
      typedef short Y;

      interface A {
            struct S {
                  struct T {
                        ArgType x[I];    // ArgType and I introduced
                        long y;          // a new y is defined, the existing Y
                                         // is not used
                  } m;
            };
            typedef string ArgType;     // Error: ArgType redefined
            enum I { I1, I2 };          // Error: I redefined
            typedef short Y;            // OK
      };    // Potential scope of  ArgType and I ends here

      interface B : A {
            typedef long ArgType  // OK, redefined in derived interface
            struct S {            // OK, redefined in derived interface
                  ArgType x;      // x is a long
            A::ArgType y;       // y is a string
        };
    };
};
```

A type may not be redefined within its scope or potential scope, as shown in the preceding example. This rule prevents type names from changing their meaning throughout a non-module scope definition, and ensures that reordering of definitions in the presence of introduced types does not affect the semantics of a specification.

Note that, in the following, the definition of **M::A::U::I** is legal because it is outside the potential scope of the I introduced in the definition of **M::A::S::T::ArgType**. However, the definition of **M::A::I** is still illegal because it is within the potential scope of the I introduced in the definition of **M::A::S::T::ArgType**.

```
module M {
    typedef long ArgType;
    const long I = 10;

    interface A {
        struct S {
            struct T {
                ArgType x[I];   // ArgType and I introduced
            } m;
        };
        struct U {
            long I;             // OK, I is not a type name
        };
        enum I { I1, I2 };      // Error: I redefined
    }; // Potential scope of  ArgType and I ends here
};
```

Note that redefinition of a type after use in a module is OK as in the example:

```
typedef long ArgType;
module M {
    struct S {
        ArgType x;          // x is a long
    };

    typedef string ArgType;     // OK!
    struct T {
        ArgType y;          // Ugly but OK, y is a string
    };
};
```

7 Repository IDs

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter.

7.1 Overview

RepositoryIds are values that can be used to establish the identity of information in the repository. A **RepositoryId** is represented as a string, allowing programs to store, copy, and compare them without regard to the structure of the value. It does not matter what format is used for any particular **RepositoryId**. However, conventions are used to manage the name space created by these IDs.

RepositoryIds may be associated with OMG IDL definitions in a variety of ways. Installation tools might generate them, they might be defined with pragmas in OMG IDL source, or they might be supplied with the package to be installed. Ensuring consistency of **RepositoryId**s with the IDL source or the IR contents is the responsibility of the programmer allocating **Repositoryid**s.

The format of the id is a short format name followed by a colon (“:”) followed by characters according to the format. This specification requires support of the OMG IDL Format. Other CORBA specifications define four formats:

1. one derived from OMG IDL names,
2. one that uses Java class names and Java serialization version UIDs,
3. one that uses DCE UUIDs, and
4. another intended for short-term use, such as in a development environment.

Since new repository ID formats may be added from time to time, compliant IDL compilers must accept any string value of the form

“<format>:<string>”

provided as the argument to the id pragma and use it as the repository ID. The OMG maintains a registry of allocated format identifiers. The **<format>** part of the ID may not contain a colon (:) character.

The version and prefix pragmas only affect default repository IDs that are generated by the IDL compiler using the IDL format.

7.2 OMG IDL Format

The OMG IDL format for **RepositoryIds** primarily uses OMG IDL scoped names to distinguish between definitions. It also includes an optional unique prefix, and major and minor version numbers.

The **RepositoryId** consists of three components, separated by colons, (“:”)

1. The first component is the format name, “IDL.”
2. (".").The second component is a list of identifiers, separated by “/” characters. These identifiers are arbitrarily long

sequences of alphabetic, digit, underscore (“_”), hyphen (“-”), and period (“.”) characters. Typically, the first identifier is a unique prefix, and the rest are the OMG IDL Identifiers that make up the scoped name of the definition. The second component shall not contain a trailing slash ("/") and it shall not begin with the characters underscore ("_"), hyphen ("-") or period (".").

3. The third component is made up of major and minor version numbers, in decimal format, separated by a “.”. When two interfaces have **RepositoryId**s differing only in minor version number it can be assumed that the definition with the higher version number is upwardly compatible with (i.e., can be treated as derived from) the one with the lower minor version number.

7.3 Pragma Directives for RepositoryId

Three pragma directives (id, prefix, and version), are specified to accommodate arbitrary **RepositoryId** formats and still support the OMG IDL **RepositoryId** format with minimal annotation. The prefix and version pragma directives apply only to the IDL format. An IDL compiler must interpret these annotations as specified. Conforming IDL compilers may support additional non-standard pragmas, but must not refuse to compile IDL source containing non-standard pragmas that are not understood by the compiler.

7.3.1 The ID Pragma

An OMG IDL pragma of the format

#pragma ID <name> “<id>”

associates an arbitrary **RepositoryId** string with a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained.The **<id>** must be a repository ID of the form described in Section 7.4, “RepositoryIDs for OMG-Specified Types,” on page 101.

An attempt to assign a repository ID to the same IDL construct a second time shall be an error unless the repository ID used in the attempt is identical to the previous one.

```
interface A {};
#pragma ID A “IDL:A:1.1”
#pragma ID A “IDL:X:1.1”       // Compile-time error

interface B {};
#pragma ID B “IDL:BB:1.1”
#pragma ID B “IDL:BB:1.1”      // OK, same ID
```

It is also an error to apply an ID to a forward-declared IDL construct (interface, valuetype, structure, and union) and then later assign a different ID to that IDL construct.

7.3.2 The Prefix Pragma

An OMG IDL pragma is of the form:

#pragma prefix “<string>”

This sets the current prefix used in generating OMG IDL format **RepositoryId**s. For example, the **RepositoryId** for the initial version of interface **Printer** defined on module **Office** by an organization known as “SoftCo” might be “IDL:SoftCo/Office/Printer:1.0.”

Since the "prefix" pragma applies to Repository Ids of the IDL format, the <string> above shall be a list of one or more identifiers, separated by the “/” characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore (“_”), hyphen (“-”), and period (“.”) characters. The string shall not contain a trailing slash ("/") and it shall not begin with the characters underscore ("_"), hyphen ("-") or period (".").

This format makes it convenient to generate and manage a set of IDs for a collection of OMG IDL definitions. The person creating the definitions sets a prefix (“SoftCo”), and the IDL compiler or other tool can synthesize all the needed IDs.

Because **RepositoryId**s may be used in many different computing environments and ORBs, as well as over a long period of time, care must be taken in choosing them. Prefixes that are distinct, such as trademarked names, domain names, UUIDs, and so forth, are preferable to generic names such as “document.”

The specified prefix applies to **RepositoryId**s generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered. An IDL file forms a scope for this purpose, so a prefix resets to the previous prefix at the end of the scope of an included file:

```
// A.idl
#pragma prefix “A”
interface A {};

// B.idl
#pragma prefix “B”
#include “A.idl”
interface B {};
```

The repository IDs for interfaces A and B in this case are:

```
IDL:A/A:1.0
IDL:B/B:1.0
```

Similarly, a prefix in an including file does not affect the prefix of an included file:

```
// C.idl
interface C {};

// D.idl
#pragma prefix “D”
#include “C.idl”
interface D {};
```

The repository IDs for interface C and D in this case are:

```
IDL:C:1.0
IDL:D/D:1.0
```

If an included file does not contain a #pragma prefix, the current prefix implicitly resets to the empty prefix:

```
// E.idl
interface E {};
```

```
    // F.idl
    module M {
      #include <E.idl>
  };
```

The repository IDs for module M and interface E in this case are:

```
    IDL:M:1.0
    IDL:E:1.0
```

If a #include directive appears at non-global scope and the included file contains a prefix pragma, the included file's prefix takes precedence, for example:

```
    // A.idl
    #pragma prefix “A”
    interface A {};

    // B.idl
    #pragma prefix “B”
    module M {
    #include “A.idl”
    };
```

The repository ID for module M and interface A in this case are:

```
    IDL:B/M:1.0
    IDL:A/A:1.0
```

Forward-declared constructs (interfaces, value types, structures, and unions) must have the same prefix in effect wherever they appear. Attempts to assign conflicting prefixes to a forward-declared construct result in a compile-time diagnostic. For example:

```
    #pragma prefix “A”
    interface A;          // Forward decl.

    #pragma prefix “B”
    interface A;          // Compile-time error

    #pragma prefix “C”
    interface A {       // Compile-time error
        void op();
    };
```

A prefix pragma of the form

```
    #pragma prefix “”
```

resets the prefix to the empty string. For example:

```
    #pragma prefix “X”
    interface X {};
    #pragma prefix “”
```

```
interface Y {};
```

The repository IDs for interface X and Y in this case are:

```
IDL:X/X:1.0
IDL:Y:1.0
```

If a specification contains both a prefix pragma and an ID or version pragma, the prefix pragma does not affect the repository ID for an ID pragma, but does affect the repository ID for a version pragma:

```
#pragma prefix “A”
interface A {};
interface B {};
interface C {};
#pragma ID B “IDL:myB:1.0”
#pragma version C 9.9
```

The repository IDs for this specification are

```
IDL:A/A:1.0
IDL:myB:1.0
IDL:A/C:9.9
```

A #pragma prefix must appear before the beginning of an IDL definition. Placing a #pragma prefix elsewhere has undefined behavior, for example:

```
interface Bar
    #pragma prefix “foo”    // Undefined behavior
    {
    // ...
};
```

7.3.3 The Version Pragma

An OMG IDL pragma of the format:

```
#pragma version <name> <major>.<minor>
```

provides the version specification used in generating an OMG IDL format **RepositoryId** for a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained. The **<major>** and **<minor>** components are decimal unsigned shorts.

If no version pragma is supplied for a definition, version 1.0 is assumed.

If an attempt is made to change the version of a repository ID that was specified with an ID pragma, a compliant compiler shall emit a diagnostic:

```
interface A {};
#pragma ID A “IDL:myA:1.1”
#pragma version A 9.9            // Compile-time error
```

An attempt to assign a version to the same IDL construct a second time shall be an error unless the version used in the attempt is identical to the existing one.

```
    interface A {};
    #pragma version A 1.1
    #pragma version A 1.1           // OK
    #pragma version A 1.2           // Error

    interface B {};
    #pragma ID B “IDL:myB:1.2”
    #pragma version B 1.2           // OK
```

7.3.4 Generation of OMG IDL - Format IDs

A definition is globally identified by an OMG IDL - format **RepositoryId** if no ID pragma is encountered for it.

The ID string shall be generated by starting with the string "IDL:". Then, if the current prefix pragma is a non-empty string, it is appended, followed by a "/" character. Next, the components of the scoped name of the definition, relative to the scope in which any prefix that applies was encountered, are appended, separated by “/” characters. Finally, a “:” and the version specification are appended.

For example, the following OMG IDL:

```
module M1 {
    typedef long T1;
    typedef long T2;
    #pragma ID T2 “DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3”
};

#pragma prefix “P1”

module M2 {
    module M3 {
        #pragma prefix “P2”
        typedef long T3;
    };
    typedef long T4;
    #pragma version T4 2.4
};
```

specifies types with the following scoped names and **RepositoryId**s:

::M1::T1	IDL:M1/T1:1.0
::M1::T2	DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3
::M2::M3::T3	IDL:P2/T3:1.0
::M2::T4	IDL:P1/M2/T4:2.4

For this scheme to provide reliable global identity, the prefixes used must be unique. Two non-colliding options are suggested: Internet domain names and DCE UUIDs.

Furthermore, in a distributed world where different entities independently evolve types, a convention must be followed to avoid the same **RepositoryId** being used for two different types. Only the entity that created the prefix has authority to create new IDs by simply incrementing the version number. Other entities must use a new prefix, even if they are only making a minor change to an existing type.

Prefix pragmas can be used to preserve the existing IDs when a module or other container is renamed or moved.

```
module M4 {
#pragma prefix “P1/M2”
    module M3 {
#pragma prefix “P2”
    typedef long T3;
    };
    typedef long T4;
#pragma version T4 2.4
};
```

This OMG IDL declares types with the same global identities as those declared in module M2 above.

See Section 7.3.2, “The Prefix Pragma,” on page 96 for further details of the effects of various prefix pragma settings on the generated **RepositoryId**s.

7.4 RepositoryIDs for OMG-Specified Types

Interoperability between implementations of official OMG specifications, including but not limited to CORBA, CORBA Services, and CORBA Facilities, depends on unambiguous specification of **RepositoryId**s for all IDL-defined types in such specifications.

All official OMG IDL files shall contain the following pragma prefix directive:

```
    #pragma prefix “omg.org”
```

unless said file already contains a pragma prefix identifying the original source of the file (e.g., “**w3c.org**”).

Revisions to existing OMG specifications must not change the definition of an existing type in any way. Two types with different repository Ids are considered different types, regardless of which part of the repository Id differs.

If an implementation must extend an OMG-specified interface, interoperability requires it to derive a new interface from the standard interface, rather than modify the standard definition.

7.5 Uniqueness Constraints on Repository IDs

```
    #pragma prefix "A"
    module M {
        // ...
    };

    #pragma prefix "B"
    module M {          // Error, inconsistent repository ID
        // ...
    };
```

This definition attempts to use the same type name M with two different repository IDs in the same compilation unit. Compilers shall issue a diagnostic for this error.

The same error can arise through inclusion of source files in the same compilation unit. For example:

```
// File1.idl
module M {
    module N {
        // ...
    };
#pragma ID N "abc"
};

// File2.idl
module M {
    module N {
        // ...
    };
};

// File3.idl
#include "File1.idl
#include "File2.idl     // Error, inconsistent repository ID
// File1.idl
 module M {
     // ...
 };

// File2.idl
#include File1.idl
#pragma prefix "X"
module M {          // Error, inconsistent repository ID
     // ...
};
```

Such errors are detectable only if they occur in a single compilation unit (or in files included in a single compilation unit); if, in different compilation units, different repository IDs are used for the same module, and these compilation units are combined into a single executable, the behavior is undefined.

8 ORB Interface

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter except 8.5, ’TypeCodes’.

8.1 Overview

This chapter introduces the operations that are implemented by the ORB core, and describes some basic ones, while providing reference to the description of the remaining operations that are described elsewhere. The **ORB** interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they may be described that way and the language binding will, for consistency, make them appear that way.

8.2 The ORB Operations

The **ORB** interface contains the operations that are available to both clients and servers. These operations do not depend on any specific object adapter or any specific object reference.

```
module CORBA {

#if ! defined(CORBA_E_MICRO)
    interface TypeCode;         // forward declaration
#endif

    typedef short PolicyErrorCode;

        // for the definition of consts see Section 7.2.2.1, “PolicyErrorCode,” on page 7

    typedef unsigned long PolicyType;

    exception PolicyError {PolicyErrorCode reason;};

    typedef string RepositoryId;
    typedef string Identifier;

    typedef unsigned short ServiceType;
    typedef unsigned long ServiceOption;
    typedef unsigned long ServiceDetailType;
    typedef CORBA::OctetSeq ServiceDetailData;
    typedef sequence<ServiceOption> ServiceOptionSeq;

    const ServiceType Security = 1;
    struct ServiceDetail {
        ServiceDetailType service_detail_type;
        ServiceDetailData service_detail;
    };

    typedef sequence<ServiceDetail> ServiceDetailSeq;

    struct ServiceInformation {
        ServiceOptionSeq service_options;
        ServiceDetailSeq service_details;
    };

#if ! defined(CORBA_E_MICRO)
    native ValueFactory;
#endif

    typedef string ORBid;

    interface ORB {

        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        ORBid id();

        string object_to_string (
            in Object           obj
        );

        Object string_to_object (
            in string           str
        );

        // Service information operations

        boolean get_service_information (
            in ServiceType service_type,
            out ServiceInformation service_information
        );

        ObjectIdList list_initial_services ();

        // Initial reference operation

        Object resolve_initial_references (
            in ObjectId identifier
        ) raises (InvalidName);

        // Thread related operations
        boolean work_pending( );
        void perform_work();
        void run();

        void shutdown(
            in boolean          wait_for_completion
        );

        void destroy();

        // Policy related operations

#if ! defined(CORBA_E_MICRO)
        Policy create_policy(
            in PolicyType       type,
            in any              val
        ) raises (PolicyError);
#endif

#if ! defined(CORBA_E_MICRO)
        // Value factory operations

        ValueFactory register_value_factory(
            in RepositoryId id,
            in ValueFactory_factory
        );

        void unregister_value_factory(in RepositoryId id);

        ValueFactory lookup_value_factory(in RepositoryId id);

        void register_initial_reference(
            in ObjectId id,
            in Object obj
        ) raises (InvalidName);
    };
#endif
};
```

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by "**CORBA::**".

The operations **object_to_string** and **string_to_object** are described in "Converting Object References to Strings" on page 106.

The **list_intial_services** and **resolve_initial_references** operations are described in Section 8.3.2, "Obtaining Initial Object References," on page 111.

The **work_pending**, **perform_work**, **shutdown, destroy**, and **run** operations are described in Section 8.2.3, "Thread-Related Operations," on page 106.

The **create_policy** operations is described in Section 7.2.2.3, "Create_policy," on page 7.

The **register_value_factory**, **unregister_value_factory** and **lookup_value_factory** operations.

The **register_initial_reference** operation.

8.2.1 ORB Identity

8.2.1.1 id

```
ORBid id();
```

The **id** operation returns the identity of the ORB. The returned **ORBid** is the string that was passed to **ORB_init** (see Section 8.3.1, “ORB Initialization,” on page 109) as the **orb_identifier** parameter when the ORB was created. If that was the empty string, the returned string is the value associated with the **-ORBid** tag in the **arg_list** parameter. Calling **id** on the default ORB returns the empty string.

8.2.2 Converting Object References to Strings

8.2.2.1 object_to_string

```
string object_to_string (
    in Object            obj
);
```

8.2.2.2 string_to_object

```
Object string_to_object (
    in string            str
);
```

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

To guarantee that an ORB will understand the string form of an object reference, that ORB’s **object_to_string** operation must be used to produce the string. For all conforming ORBs, if **obj** is a valid reference to an object, then **string_to_object(object_to_string(obj))** will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB’s supporting IOP, this remains true even if the two operations are performed on different ORBs.

8.2.3 Thread-Related Operations

To support single-threaded ORBs, as well as multi-threaded ORBs that run multi-thread-unaware code, several operations are included in the **ORB** interface. These operations can be used by single-threaded and multi-threaded applications. An application that is a pure ORB client would not need to use these operations. Both the **ORB::run** and **ORB::shutdown** are useful in fully multi-threaded programs.

These operations are defined on the ORB rather than on an object adapter to allow the main thread to be used for all kinds of asynchronous processing by the ORB. Defining these operations on the ORB also allows the ORB to support multiple object adapters, without requiring the application main to know about all the object adapters. The interface between the ORB and an object adapter is not standardized.

8.2.3.1 work_pending

boolean work_pending( );

This operation returns an indication of whether the ORB needs the main thread to perform some work.

A result of TRUE indicates that the ORB needs the main thread to perform some work and a result of FALSE indicates that the ORB does not need the main thread.

8.2.3.2 perform_work

void perform_work();

If called by the main thread, this operation performs an implementation-defined unit of work; otherwise, it does nothing.

It is platform-specific how the application and ORB arrange to use compatible threading primitives.

The **work_pending()** and **perform_work()** operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multi-threaded server would need a polling loop only if there were both ORB and other code that required use of the main thread.

Here is an example of such a polling loop:

```
// C++
for (;;) {
        if (orb->work_pending()) {
                orb->perform_work();
        };
        // do other things
        // sleep?
};
```

Once the ORB has shutdown, **work_pending** and **perform_work** will raise the BAD_INV_ORDER exception with minor code 4. An application can detect this exception to determine when to terminate a polling loop.

8.2.3.3 run

void run();

This operation provides execution resources to the ORB so that it can perform its internal functions. Single threaded ORB implementations, and some multi-threaded ORB implementations, need the use of the main thread in order to function properly. For maximum portability, an application should call either **run** or **perform_work** on its main thread. **run** may be called by multiple threads simultaneously.

This operation will block until the ORB has completed the shutdown process, initiated when some thread calls **shutdown**.

8.2.3.4 shutdown

```
void shutdown(
    in boolean          wait_for_completion
);
```

This operation instructs the ORB to shut down, that is, to stop processing in preparation for destruction.

Shutting down the ORB causes all object adapters to be destroyed, since they cannot exist in the absence of an ORB.

In the case of the **POA**, all **POAManager**s are deactivated prior to destruction of all POAs. The deactivation that the ORB performs should be the equivalent of calling deactivate with the value **TRUE** for **etherealize_objects** and with the **wait_for_completion** parameter same as what **shutdown** was called with.

Shut down is complete when all **ORB** processing has completed and the object adapters have been destroyed. **ORB** processing is defined as including request processing and object deactivation or other operations associated with object adapters, and the forwarding of the responses from deferred synchronous invocations to their associated reply handlers. In the case of the **POA**, this means that all object etherealizations have finished and root POA has been destroyed (implying that all descendent **POA**s have also been destroyed).

If the **wait_for_completion** parameter is **TRUE**, this operation blocks until the shut down is complete. If an application does this in a thread that is currently servicing an invocation, the ORB will not shutdown, and the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, and completion status COMPLETED_NO, since blocking would result in a deadlock.

If the **wait_for_completion** parameter is **FALSE**, then **shutdown** may not have completed upon return. An ORB implementation may require the application to call (or have a pending call to) **run** or **perform_work** after **shutdown** has been called with its parameter set to **FALSE**, in order to complete the shutdown process.

Additionally in systems that have Portable Object Adapters (see Chapter 11) **shutdown** behaves as if **POA::destroy** is called on the Root **POA** with its first parameter set to TRUE and the second parameter set to the value of the **wait_for_completion** parameter that **shutdown** is invoked with.

While the ORB is in the process of shutting down, the ORB operates as normal, servicing incoming and outgoing requests until all requests have been completed. An implementation may impose a time limit for requests to complete while a **shutdown** is pending.

Once an ORB has shutdown, only object reference management operations(**duplicate**, **release** and **is_nil**) may be invoked on the ORB or any object reference obtained from it. An application may also invoke the destroy operation on the ORB itself. Invoking any other operation will raise the BAD_INV_ORDER system exception with the OMG minor code 4.

8.2.3.5 destroy

```
void destroy();
```

This operation destroys the ORB so that its resources can be reclaimed by the application. Any operation invoked on a destroyed ORB reference will raise the OBJECT_NOT_EXIST exception. Once an ORB has been destroyed, another call to **ORB_init** with the same **ORBid** will return a reference to a newly constructed ORB.

If **destroy** is called on an ORB that has not been shut down, it will start the shut down process and block until the ORB has shut down before it destroys the ORB. The behavior is similar to that achieved by calling **shutdown** with the **wait_for_completion** parameter set to **TRUE**. If an application calls **destroy** in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

For maximum portability and to avoid resource leaks, an application should always call **shutdown** and **destroy** on all ORB instances before exiting.

8.3 ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and possibly the object adapter (POA) environments.
- Get references to ORB pseudo-object (for use in future ORB operations) and perhaps other objects (including the root POA or some Object Adapter objects).

The following operations are provided to initialize applications and obtain the appropriate object references:

- Operations providing access to the ORB. These operations reside in the CORBA module, but not in the ORB interface and are described in Section 8.3.1, “ORB Initialization,” on page 109.
- Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Section 8.3.2, “Obtaining Initial Object References,” on page 111.

8.3.1 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get the ORB pseudo-object reference and possibly an OA object reference (such as the root POA). This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB pseudo-object reference and the OA object reference to the application for use in future ORB and OA operations.

The ORB and OA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB. The operation to initialize an application in the ORB and get its pseudo-object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application’s bootstrap call into the CORBA world. The **ORB_init** call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain other references for that ORB.

In order to obtain an **ORB** pseudo-object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo-object reference is required, and an **arg_list**, which is used to allow environment-specific data to be passed into the call. PIDL for the ORB initialization is as follows:

```
// PIDL
module CORBA {
    typedef sequence <string> arg_list;
    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

The identifier for the ORB will be a name of type **CORBA::ORBid**. All **ORBid** strings other than the empty string are allocated by ORB administrators and are not managed by the OMG. ORB administration is the responsibility of each ORB supplier. ORB suppliers may optionally delegate this responsibility. **ORBid** strings other than the empty string are intended to be used to uniquely identify each ORB used within the same address space in a multi-ORB application. These special **ORBid** strings are specific to each ORB implementation and the ORB administrator is responsible for ensuring that the names are unambiguous.

If an empty **ORBid** string is passed to **ORB_init**, then the **arg_list** arguments shall be examined to determine if they indicate an ORB reference that should be returned. This is achieved by searching the **arg_list** parameters for one preceded by "**-ORBid**" for example, "**-ORBid example_orb**" (the white space after the "**-ORBid**" tag is ignored) or "**-ORBidMyFavoriteORB**" (with no white space following the "**-ORBid**" tag). Alternatively, two sequential parameters with the first being the string "**-ORBid**" indicates that the second is to be treated as an **ORBid** parameter. If an empty string is passed and no **arg_list** parameters indicate the ORB reference to be returned, the default ORB for the environment will be returned.

Other parameters of significance to the ORB can also be identified in **arg_list**, for example, "**Hostname**," "**SpawnedServer**," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the parameter format that ORB parameters may take. In general, parameters shall be formatted as either one single **arg_list** parameter:

–ORB<suffix><optional white space> <value>

or as two sequential **arg_list** parameters:

-ORB<suffix>

<value>

Regardless of whether an empty or non-empty **ORBid** string is passed to **ORB_init**, the **arg_list** arguments are examined to determine if any ORB parameters are given. If a non-empty **ORBid** string is passed to **ORB_init**, all **ORBid** parameters in the **arg_list** are ignored. All other **-ORB<suffix>** parameters in the **arg_list** may be of significance during the ORB initialization process.

Before **ORB_init** returns, it will remove from the **arg_list** parameter all strings that match the **-ORB<suffix>** pattern described above and that are recognized by that ORB implementation, along with any associated sequential parameter strings. If any strings in **arg_list** that match this pattern are not recognized by the ORB implementation, **ORB_init** will raise the BAD_PARAM system exception instead.

The **ORB_init** operation may be called any number of times and shall return the same ORB reference when the same **ORBid** string is passed, either explicitly as an argument to **ORB_init** or through the **arg_list**. All other **-ORB<suffix>** parameters in the **arg_list** may be considered on subsequent calls to **ORB_init**.

Note – Whenever an ORB_init argument of the form -ORBxxx is specified, it is understood that the argument may be represented in different ways in different languages. For example, in Java -ORBxxx is equivalent to a property named org.omg.CORBA.ORBxxx.

8.3.1.1 Server ID

A Server ID must uniquely identify a server to an IMR. This specification only requires unique identification using a string of some kind. We do not intend to make more specific requirements for the structure of a server ID.

The server ID may be specified by an **ORB_init** argument of the form

-ORBServerId

The value assigned to this property is a **string**. All templates created in this **ORB** will return this server ID in the **server_id** attribute.

It is required that all ORBs in the same server share the same server ID. Specific environments may choose to implement **-ORBServerId** in ways that automatically enforce this requirement.

For example, the **org.omg.CORBA.ServerId** system property may be set to the server ID in Java when a Java server is activated. This system property is then picked up as part of the **ORB_init** call for every **ORB** created in the server.

8.3.1.2 Server Endpoint

The server endpoint information is passed into **ORB_init** by an argument of the form

-ORBListenEndpoints <endpoints>

The format of the <**endpoints**> argument is proprietary. All that is required by this specification is that each time **ORB_init** is called with the same value for this argument, the resulting **ORB** will listen for requests on the same set of endpoints, so that persistent object references for the **ORB** will continue to function correctly.

8.3.1.3 Starting Servers with No Proprietary Server Activation Support

Any server started with the flag:

-ORBNoProprietaryActivation

shall avoid the use of any proprietary activation framework.

8.3.2 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the root POA, POA Current, Interface Repository and various Object Services instances. (The POA is described in the Portable Object Adapter chapter; the Interface Repository is described in the Interface Repository chapter; Object Services are described in the individual service specifications.) The functionality required by the application is similar to that provided by the Naming Service. However, the OMG does not want to mandate that the Naming Service be made available to all applications in order that they may be portably initialized. Consequently, the operations shown in this section provide a simplified, local version of the Naming Service that applications can use to obtain a small, defined set of object references that are essential to its operation. Because only a small well-defined set of objects are expected with this mechanism, the naming context can be flattened to be a single-level name space. This simplification results in only two operations being defined to achieve the functionality required.

Initial references are not obtained via a new interface; instead two operations are provided in the ORB pseudo-object interface, providing facilities to list and resolve initial object references.

8.3.2.1 list_initial_services

```
typedef string ObjectId;
typedef sequence <ObjectId> ObjectIdList;
ObjectIdList list_initial_services ();
```

8.3.2.2 resolve_initial_references

```
exception InvalidName {};

Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);
```

The **resolve_initial_references** operation is an operation on the ORB rather than the Naming Service's **NamingContext**. The interface differs from the Naming Service's resolve in that **ObjectId** (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the name space to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this route. Unlike the ORB identifiers, the **ObjectId** name space requires careful management. To achieve this, the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

resolve_initial_references never returns a **nil** reference. Instead, the non-availability of a particular reference is indicated by throwing an **InvalidName** exception (even if a **nil** reference is explicitly configured for an **ObjectId**).

Currently, reserved **ObjectIds** are **RootPOA**, **POACurrent**, **InterfaceRepository, NameService**, **TradingService**, **SecurityCurrent**, **TransactionCurrent, DynAnyFactory, ORBPolicyManager, PolicyCurrent, NotificationService, TypedNotificationService, CodecFactory, PICurrent, ComponentHomeFinder**, and **PSS.**.

Table 8.1- ObjectIds for resolve_initial_references

ObjectId	Type of Object Reference	Reference
RootPOA	PortableServer::POA	Section 8.3.4, "POA Interface," on page 17
POACurrent	PortableServer::Current	Section 8.3.4, "POA Interface," on page 17
InterfaceRepository	CORBA::Repository CORBA::ComponentIR::Repository	Section 10.5.6, "Repository" on page 10-22 in formal/04-03-01. Section 10.6.2, "ComponentIR::Repository" on page 10-52 in formal/04-03-01.
NameService	CosNaming:: NamingContext	Naming Service specification (formal/04-10-03), the CosNaming Module section.

Table 8.1- ObjectIds for resolve_initial_references

ObjectId	Type of Object Reference	Reference
TradingService	CosTrading::Lookup	Trading Object Service specification (formal/00-06-27), the Functional Interfaces section.
SecurityCurrent	SecurityLevel1::Current or SecurityLevel2::Current	Security Service specification (formal/00-06-25), the Security Operations on Current section.
TransactionCurrent	CosTransaction::Current	Transaction Service specification (formal/00-06-28), the Transaction Service Interfaces section.
DynAnyFactory	DynamicAny:: DynAnyFactory	Section 9.2.1, “Creating a DynAny Object” in formal 04-03-01.
ORBPolicyManager	CORBA::PolicyManager	Section 11.2.3, “Policy Management Interfaces” in formal/ 04-03-01.
PolicyCurrent	CORBA::PolicyCurrent	Section 7.3.3, “Policy Management Interfaces,” on page 9.
NotificationService	CosNotifyChannelAdmin:: EventChannelFactory	Notification Service specification (formal/00-06-20)
TypedNotificationService	CosTypedNotifyChannelAdmin::TypedEvent ChannelFactory	Notification Service specification (formal/00-06-20)
CodecFactory	IOP::CodecFactory	Section 14.1.2, “Codec Factory” in formal/04-03-01.
PICurrent	PortableInterceptors::Current	Section 13.5.3, “Portable Interceptor Current Interface” in formal/04-03-01.
ComponentHomeFinder	Components::HomeFinder	*CORBA Components* specification.
PSS	CosPersistentState:: ConnectorRegistry	*Persistent State* specification (ptc/ 01-12-02).

To allow an application to determine which objects have references available via the initial references mechanism, the **list_initial_services** operation (also a call on the ORB) is provided. It returns an **ObjectIdList,** which is a sequence of **ObjectIds**. **ObjectIds** are typed as strings. Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the id, the type of object being returned must be defined; that is, “**InterfaceRepository**” returns an object of type **Repository**, or **ComponentIR::Repository**, which is derived from **Repository**, depending on whether the ORB supports components or not, and “**NameService**” returns a **CosNaming::NamingContext** object.

The application is responsible for narrowing the object reference returned from **resolve_initial_references** to the type that was requested in the ObjectId. For example, for **InterfaceRepository** the object returned would be narrowed to **Repository** type or **ComponentIR::Repository** type, depending on whether the ORB supports components.

Specifications for Object Services (see individual service specifications) state whether it is expected that a service's initial reference be made available via the **resolve_initial_references** operation or not; that is, whether the service is necessary or desirable for bootstrap purposes.

8.3.3 Configuring Initial Service References

8.3.3.1 ORB-specific Configuration

It is required that an ORB can be administratively configured to return an arbitrary object reference from **CORBA::ORB::resolve_initial_references** for non-locality-constrained objects.

In addition to this required implementation-specific configuration, two **CORBA::ORB_init** arguments are provided to override the ORB initial reference configuration.

8.3.3.2 ORBInitRef

The ORB initial reference argument, **-ORBInitRef**, allows specification of an arbitrary object reference for an initial service. The format is:

-ORBInitRef <ObjectID>=<ObjectURL>

Examples of use are:

-ORBInitRef NameService=IOR:00230021AB...

-ORBInitRef NotificationService=corbaloc::555objs.com/NotificationService

-ORBInitRef TradingService=corbaname::555objs.com#Dev/Trader

<ObjectID> represents the well-known **ObjectID** for a service defined in the CORBA specification, such as **NameService**. This mechanism allows an ORB to be configured with new initial service Object IDs that were not defined when the ORB was installed.

<ObjectURL> can be any of the URL schemes supported by **CORBA::ORB::string_to_object**, with the exception of the corbaloc URL scheme with the rir protocol (i.e., corbaloc:rir...). If a URL is syntactically malformed or can be determined to be invalid in an implementation defined manner, **ORB_init** raises a **BAD_PARAM** exception.

8.3.3.3 ORBDefaultInitRef

The ORB default initial reference argument, **-ORBDefaultInitRef**, assists in resolution of initial references not explicitly specified with `-ORBInitRef`. **-ORBDefaultInitRef** requires a URL that, after appending a slash '/' character and a stringified object key, forms a new URL to identify an initial object reference. For example:

-ORBDefaultInitRef corbaloc::555objs.com

A call to **resolve_initial_references (**see the "NotificationService") with this argument results in a new URL:

corbaloc::555objs.com/NotificationService

That URL is passed to **CORBA::ORB::string_to_object** to obtain the initial reference for the service.

Another example is:

**-ORBDefaultInitRef **
corbaname::555ResolveRefs.com,:555Backup.com#Prod/Local

After calling **resolve_initial_references("NameService")**, one of the `corbaname` URLs

corbaname::555ResolveRefs.com#Prod/Local/NameService

or

corbaname::555Backup411.com#Prod/Local/NameService

is used to obtain an object reference from **string_to_object**. (In this example, **Prod/Local/NameService** represents a stringified **CosNaming::Name**).

The **-ORBDefaultInitRef** argument naturally extends to URL schemes that may be defined in the future, provided the final part of the URL is an object key.

8.3.3.4 Configuration Effect on resolve_initial_references

Default Resolution Order

The default order for processing a call to **CORBA::ORB::resolve_initial_references** for a given **<ObjectID>** is:

1. Resolve with **register_initial_reference** entry if possible.
2. Resolve with **-ORBInitRef** for this **<ObjectID>** if possible
3. Resolve with pre-configured ORB settings if possible.
4. Resolve with an **-ORBDefaultInitRef** entry if possible.

ORB Configured Resolution Order

There are cases where the default resolution order may not be appropriate for all services and use of **-ORBDefaultInitRef** may have unintended resolution side effects. For example, an ORB may use a proprietary service, such as **ImplementationRepository**, for internal purposes and may want to prevent a client from unknowingly diverting the ORB's reference to an implementation repository from another vendor. To prevent this, an ORB is allowed to ignore the **-ORBDefaultInitRef** argument for any or all **<ObjectID>**s for those services that are not OMG-specified services with a well-known service name as accepted by **resolve_initial_references**. An ORB can only ignore the **-ORBDefaultInitRef** argument but must always honor the **-ORBInitRef** argument.

8.3.3.5 Configuration Effect on list_initial_services

The **<ObjectID>**s of all **-ORBInitRef argument**s to **ORB_init** appear in the list of tokens returned by **list_initial_services** as well as all ORB-configured **<ObjectID>**s. Any other tokens that may appear are implementation-dependent.

The list of **<ObjectID>**s returned by **list_initial_services** can be a subset of the **<ObjectID>**s recognized as valid by **resolve_initial_references**.

8.4 Current Object

ORB and CORBA services may wish to provide access to information (context) associated with the thread of execution in which they are running. This information is accessed in a structured manner using interfaces derived from the **Current** interface defined in the **CORBA** module.

Each ORB or CORBA service that needs its own context derives an interface from the **CORBA** module's **Current**. Users of the service can obtain an instance of the appropriate **Current** interface by invoking **ORB::resolve_initial_references**. For example the Security service obtains the **Current** relevant to it by invoking

```
    ORB::resolve_initial_references("SecurityCurrent")
```

A CORBA service does not have to use this method of keeping context but may choose to do so.

```
module CORBA {
    // interface for the Current object
     local interface Current {
    };
};
```

Operations on interfaces derived from **Current** access state associated with the thread in which they are invoked, not state associated with the thread from which the **Current** was obtained. This prevents one thread from manipulating another thread's state, and avoids the need to obtain and narrow a new **Current** in each method's thread context.

Current objects must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. **Current**s are per-process singleton objects, so no destroy operation is needed.

8.5 TypeCodes

TypeCodes are values that represent invocation argument types and attribute types. They can be obtained from IDL compilers.

TypeCodes have a number of uses. They are crucial to the semantics of the **any** type.

Abstractly, **TypeCode**s consist of a "kind" field, and a set of parameters appropriate for that kind. For example, the **TypeCode** describing OMG IDL type **long** has kind **tk_long** and no parameters. The **TypeCode** describing OMG IDL type **sequence<boolean,10>** has kind **tk_sequence** and two parameters: **10** and **boolean**.

8.5.1 The TypeCode Interface

The PIDL interface for **TypeCodes** is as follows:

```
module CORBA {
    enum TCKind {
        tk_null, tk_void,
        tk_short, tk_long, tk_ushort, tk_ulong,
        tk_float, tk_double, tk_boolean, tk_char,
        tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
        tk_struct, tk_union, tk_enum, tk_string,
        tk_sequence, tk_array, tk_alias, tk_except,
        tk_longlong, tk_ulonglong, tk_longdouble,
        tk_wchar, tk_wstring, tk_fixed,
        tk_value, tk_value_box,
        tk_native,
        tk_abstract_interface,
        tk_local_interface
        tk_component, tk_home,
        tk_event
    };

    typedef short ValueModifier;
        const ValueModifier VM_NONE = 0;
        const ValueModifier VM_CUSTOM = 1;
        const ValueModifier VM_ABSTRACT = 2;
        const ValueModifier VM_TRUNCATABLE = 3;

    interface TypeCode {
        exception       Bounds {};
        exception       BadKind {};

        // for all TypeCode kinds
        boolean equal (in TypeCode tc);

        boolean equivalent(in TypeCode tc);
        TypeCode get_compact_typecode();

        TCKind kind ();
        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // tk_local_interface, tk_except
        // tk_component, tk_home and tk_event
        RepositoryId id () raises (BadKind);

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // tk_local_interface, tk_except
        // tk_component, tk_home and tk_event
        Identifier name () raises (BadKind);

        // for tk_struct, tk_union, tk_enum, tk_value,
        // tk_except and tk_event
        unsigned long member_count () raises (BadKind);
        Identifier member_name (in unsigned long index)
            raises(BadKind, Bounds);

        // for tk_struct, tk_union, tk_value,
        // tk_except and tk_event
```

```
        TypeCode member_type (in unsigned long index)
            raises (BadKind, Bounds);

        // for tk_union
        any member_label (in unsigned long index)
            raises(BadKind, Bounds);
        TypeCode discriminator_type () raises (BadKind);
        long default_index () raises (BadKind);

        // for tk_string, tk_wstring, tk_sequence, and tk_array
        unsigned long length () raises (BadKind);

        // for tk_sequence, tk_array, tk_value_box and tk_alias
        TypeCode content_type () raises (BadKind);

        // for tk_fixed
        unsigned short fixed_digits() raises(BadKind);
        short fixed_scale() raises(BadKind);

        // for tk_value and tk_event
        Visibility member_visibility(in unsigned long index)
            raises(BadKind, Bounds);
        ValueModifier type_modifier() raises(BadKind);
        TypeCode concrete_base_type() raises(BadKind);
    };
};
```

With the above operations, any **TypeCode** can be decomposed into its constituent parts. The BadKind exception is raised if an operation is not appropriate for the **TypeCode** kind it invoked.

The **equal** operation can be invoked on any **TypeCode**. The **equal** operation returns **TRUE** if and only if for the target **TypeCode** and the **TypeCode** passed through the parameter **tc,** the set of legal operations is the same and invoking any operation from that set on the two **TypeCode**s return identical results.

The **equivalent** operation is used by the ORB when determining type equivalence for values stored in an IDL **any**. **TypeCodes** are considered equivalent based on the following semantics:

- If the result of the **kind** operation on either **TypeCode** is **tk_alias**, recursively replace the **TypeCode** with the result of calling **content_type**, until the kind is no longer **tk_alias**.
- If results of the **kind** operation on each typecode differ, **equivalent** returns false.
- If the **id** operation is valid for the **TypeCode kind**, **equivalent** returns **TRUE** if the results of **id** for both **TypeCodes** are non-empty strings and both strings are equal. If both ids are non-empty but are not equal, then **equivalent** returns **FALSE**. If either or both id is an empty string, or the **TypeCode kind** does not support the **id** operation, **equivalent** will perform a structural comparison of the **TypeCodes** by comparing the results of the other **TypeCode** operations in the following bullet items (ignoring aliases as described in the first bullet.). The structural comparison only calls operations that are valid for the given **TypeCode kind**. If any of these operations do not return equal results, then **equivalent** returns **FALSE**. If all comparisons are equal, **equivalent** returns true.
- The results of the **name** and **member_name** operations are ignored and not compared.

- The results of the **member_count**, **default_index**, **length**, **digits**, **scale,** and **type_modifier** operations are compared.
- The results of the **member_label** operation for each member index of a **union TypeCode** are compared for equality. Note that this means that **unions** whose members are not defined in the same order are not considered structurally equivalent.
- The results of the **discriminator_type, member_type,** and **concrete_base_type** operation and for each member index, and the result of the **content_type** operation are compared by recursively calling **equivalent**.
- The results of the **member_visibility** operation are compared for each member index.

Applications that need to distinguish between a type and different aliases of that type can supplement **equivalent** by directly invoking the **id** operation and comparing the results.

The **get_compact_typecode** operation strips out all optional **name** and **member name** fields, but it leaves all alias typecodes intact.

The **kind** operation can be invoked on any **TypeCode**. Its result determines what other operations can be invoked on the **TypeCode**.

The **id** operation can be invoked on object reference, valuetype, boxed valuetype, abstract interface, local interface, native, structure, union, enumeration, alias, exception, component, home, and event **TypeCode**s. It returns the **RepositoryId** globally identifying the type. Object reference, valuetype, boxed valuetype, native, exception, component, home, and event **TypeCode**s always have a **RepositoryId**. Structure, union, enumeration, and alias **TypeCode**s obtained from the Interface Repository or the **ORB::create_operation_list** operation also always have a **RepositoryId**. Otherwise, the **id** operation can return an empty string.

When the **id** operation is invoked on an object reference **TypeCode** that contains a base **Object**, the returned value is **IDL:omg.org/CORBA/Object:1.0**.

When it is invoked on a valuetype **TypeCode** that contains a **ValueBase**, the returned value is **IDL:omg.org/CORBA/ValueBase:1.0**.

The **name** operation can also be invoked on object reference, structure, union, enumeration, alias, abstract interface, local interface, value type, boxed valuetype, native, and exception **TypeCode**s. It returns the simple name identifying the type within its enclosing scope. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the type in any particular **Repository**, and may even be an empty string.

The order in which members are presented in the interface repository is the same as the order in which they appeared in the IDL specification, and this ordering determines the index value for each member. The first member has index value 0. For example for a structure definition:

```
struct example {
    short   member1;
    short   member2;
    long    member3;
};
```

In this example **member1** has **index** = 0, **member2** has **index** = 1, and **member3** has **index** = 2. The value of **member_count** in this case is 3.

The **member_count** and **member_name** operations can be invoked on structure, union, non-boxed valuetype, non-boxed eventtype, exception, and enumeration **TypeCode**s. **Member_count** returns the number of members constituting the type. **Member_name** returns the simple name of the member identified by **index**. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the member in any particular **Repository**, and may even be an empty string.

The **member_type** operation can be invoked on structure, non-boxed valuetype, non-boxed eventtype, exception and union **TypeCode**s. It returns the **TypeCode** describing the type of the member identified by **index**.

The **member_label**, **discriminator_type**, and **default_index** operations can only be invoked on union **TypeCode**s. **Member_label** returns the label of the union member identified by **index**. For the default member, the label is the zero octet. The **discriminator_type** operation returns the type of all non-default member labels. The **default_index** operation returns the index of the default member, or -1 if there is no default member.

The **member_visibility** operation can only be invoked on non-boxed valuetype and non-boxed eventtype, **TypeCodes**. It returns the **Visibility** of the valuetype/eventtype member identified by index.

The **member_name**, **member_type**, **member_label**, and **member_visibility** operations raise **Bounds** if the index parameter is greater than or equal to the number of members constituting the type.

The **content_type** operation can be invoked on sequence, array, boxed valuetype and alias **TypeCode**s. For sequences and arrays, it returns the element type. For aliases, it returns the original type. For boxed valuetype, it returns the boxed type.

An array **TypeCode** only describes a single dimension of an OMG IDL array. Multi-dimensional arrays are represented by nesting **TypeCode**s, one per dimension. The outermost **tk_array Typecode** describes the leftmost array index of the array as defined in IDL. Its **content_type** describes the next index. The innermost nested **tk_array TypeCode** describes the rightmost index and the array element type.

The **type_modifier** and **concrete_base_type** operations can be invoked on non-boxed valuetype and non-boxed eventtype**TypeCode**s. The **type_modifier** operation returns the **ValueModifier** that applies to the valuetype/eventtype represented by the target **TypeCode**. If the valuetype/eventtype represented by the target **TypeCode** has a concrete base valuetype/eventtype, the **concrete_base_type** operation returns a **TypeCode** for the concrete base, otherwise it returns a nil **TypeCode** reference.

The **length** operation can be invoked on string, wide string, sequence, and array **TypeCode**s. For strings and sequences, it returns the bound, with zero indicating an unbounded string or sequence. For arrays, it returns the number of elements in the array. For wide strings, it returns the bound, or zero for unbounded wide strings.

8.5.2 TypeCode Constants

For IDL type declarations, the IDL compiler produces (if asked) a declaration of a **TypeCode** constant. See the language mapping rules for more information about the names of the generated **TypeCode** constants. **TypeCode** constants include tk_alias definitions wherever an IDL typedef is referenced. These constants can be used with the dynamic invocation interface and other routines that require **TypeCode**s.

The predefined **TypeCode** constants, named according to the C language mapping, are:

TC_null
TC_void
TC_short
TC_long

TC_longlong
TC_ushort
TC_ulong
TC_ulonglong
TC_float
TC_double
TC_longdouble
TC_boolean
TC_char
TC_wchar
TC_octet
TC_any
TC_TypeCode
TC_Object	= tk_objref {Object}
TC_string	= tk_string {0} // unbounded
TC_wstring = tk_wstring{0}	/// unbounded
TC_ValueBase = tk_value {ValueBase}
TC_Component = tk_component {CCMObject}
TC_Home = tk_home {CCMHome}
TC_EventBase = tk_event {EventBase}

For the **TC_Object TypeCode** constant, calling **id** returns "**IDL:omg.org/CORBA/Object:1.0**" and calling **name** returns "**Object.**"

For the **TC_ValueBase TypeCode** constant, calling **id** returns "**IDL:omg.org/CORBA/ValueBase:1.0,**" calling **name** returns "**ValueBase,**" calling **member_count** returns **0**, calling **type_modifier** returns **CORBA::VM_NONE**, and calling **concrete_base_type** returns a **nil TypeCode**.

For the **TC_Component TypeCode** constant, calling **id** returns "**IDL:omg.org/Components/CCMObject:1.0**" and calling **name** returns "**CCMObject**."

For the **TC_Home TypeCode** constant, calling **id** returns "**IDL:omg.org/Components/CCMHome:1.0**" and calling **name** returns "**CCMHome**."

For the **TC_EventBase TypeCode** constant, calling **id** returns "**IDL:omg.org/Components/EventBase:1.0**," calling **name** returns "**EventBase**," calling **member_count** returns **0**, calling **type_modifier** returns **CORBA::VM_NONE**, and calling **concrete_base_type** returns a **nil TypeCode**.

8.6 Exceptions

The terms “system” and “user” exception are defined in this section. Further the terms “standard system exception” and “standard user exception” are defined, and then a list of “standard system exceptions” is provided.

8.6.1 Definition of Terms

In general the following terms should be used consistently in all OMG standards documents to refer to exceptions:

Standard Exception: Any exception that is defined in an OMG Standard.

System Exception: Clients must be prepared to handle these exceptions even though they are not declared in a raises clause. These exceptions cannot appear in a raises clause. These have the structure defined in section 3.17.2 “System Exception,” and they are of type **SYSTEM_EXCEPTION** (see PIDL below).

Standard System Exception: A System Exception that is part of the CORBA Standard as enumerated in section 3.17. (e.g., BAD_PARAM). These are enumerated in Section 3.17.2 “Standard System Exceptions.”

Non-Standard System Exceptions: System exceptions that are proprietary to a particular vendor/implementation.

User Exception: Exceptions that can be raised only by those operations that explicitly declare them in the raises clause of their signature. These exceptions are of type USER_EXCEPTION (see IDL below).

Standard User Exception: Any User Exception that is defined in a published OMG standard (e.g., WrongTransaction). These are documented in the documentation of individual interfaces.

Non-standard User Exception: User exceptions that are not defined in any published OMG specification.

8.6.2 System Exceptions

In order to bound the complexity in handling the standard exceptions, the set of standard exceptions should be kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions corresponding to the different ways that memory allocation failure causes the exception (during marshaling, unmarshaling, in the client, in the object implementation, allocating network packets), a single exception corresponding to dynamic memory allocation failure is defined.

```
module CORBA {
    const unsigned long OMGVMCID = 0x4f4d0000;

#define ex_body {unsigned long minor; completion_status completed;}

    enum completion_status {
        COMPLETED_YES,
        COMPLETED_NO,
        COMPLETED_MAYBE
    };

    enum exception_type {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION
    };
};
```

Each system exception includes a minor code to designate the subcategory of the exception.

Minor exception codes are of type **unsigned long** and consist of a 20-bit “Vendor Minor Codeset ID”(**VMCID**), which occupies the high order 20 bits, and the minor code which occupies the low order 12 bits.

The standard minor codes for the standard system exceptions are prefaced by the **VMCID** assigned to OMG, defined as the unsigned long constant **CORBA::OMGVMCID**, which has the VMCID allocated to OMG occupying the high order 20 bits. The minor exception codes associated with the standard exceptions that are found in Annex A are or-ed with **OMGVMCID** to get the minor code value that is returned in the **ex_body** structure (see Section 8.6.3, "Standard System Exception Definitions," on page 123 and Section 8.6.4, "Standard Minor Exception Codes," on page 130).

Within a vendor assigned space, the assignment of values to minor codes is left to the vendor. Vendors may request allocation of **VMCID**s by sending email to *tag-request@omg.org*.

The **VMCID** *0* and *0xfffff* are reserved for experimental use. The **VMCID OMGVMCID** (Section 8.6.3, "Standard System Exception Definitions," on page 123) and *1* through *0xf* are reserved for OMG use.

Each standard system exception also includes a **completion_status** code that takes one of the values {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

COMPLETED_YES	The object implementation has completed processing prior to the exception being raised.
COMPLETED_NO	The object implementation was never initiated prior to the exception being raised.
COMPLETED_MAYBE	The status of implementation completion is indeterminate.

Client applications must be prepared to handle system exceptions other than the standard system exception defined in Section 8.6.3, "Standard System Exception Definitions," on page 123, both because future versions of this specification may define additional standard system exceptions, and because ORB implementations may raise non-standard system exceptions.

Vendors may define non-standard system exceptions, but these exceptions are discouraged because they are non-portable. A non-standard system exception, when passed to an ORB that does not recognize it, shall be presented by that ORB as an **UNKNOWN** standard system exception. The completion status shall be preserved in the **UNKNOWN** exception, and the minor code shall be set to standard value 2 for system exception and standard value 1 for user exception.

Non-standard system exceptions shall have the same structure as of standard standard system exceptions as specified in section Section 8.6.3, "Standard System Exception Definitions," on page 123 (i.e., they have the same ex_body). They also shall follow the same language mappings as standard system exceptions. Although they are PIDL, vendors should ensure that their names do not clash with any other names following the normal naming and scoping rules as they apply to regular IDL exceptions.

8.6.3 Standard System Exception Definitions

The standard system exceptions are defined in this section.

```
module CORBA {                  // PIDL

    exception UNKNOWN ex_body;
                                    // the unknown exception
    exception BAD_PARAM ex_body;
                                    // an invalid parameter was passed
    exception NO_MEMORY ex_body;
                                    // dynamic memory allocation failure
    exception IMP_LIMIT ex_body;
                                    // violated implementation limit
```

```
exception COMM_FAILURE ex_body;
                             // communication failure
exception INV_OBJREF ex_body;
                             // invalid object reference
exception NO_PERMISSION ex_body;
                             // no permission for attempted op.
exception INTERNAL ex_body;
                             // ORB internal error
exception MARSHAL ex_body;
                             // error marshaling param/result
exception INITIALIZE ex_body;
                             // ORB initialization failure
exception NO_IMPLEMENT ex_body;
                             // operation implementation unavailable
exception BAD_TYPECODE ex_body;
                             // bad typecode
exception BAD_OPERATION ex_body;
                             // invalid operation
exception NO_RESOURCES ex_body;
                             // insufficient resources for req.
exception NO_RESPONSE ex_body;
                             // response to req. not yet available
exception PERSIST_STORE ex_body;
                             // persistent storage failure
exception BAD_INV_ORDER ex_body;
                             // routine invocations out of order
exception TRANSIENT ex_body;
                             // transient failure - reissue request
exception FREE_MEM ex_body;
                             // cannot free memory
exception INV_IDENT ex_body;
                             // invalid identifier syntax
exception INV_FLAG ex_body;
                             // invalid flag was specified
exception INTF_REPOS ex_body;
                             // error accessing interface repository
exception BAD_CONTEXT ex_body;
                             // error processing context object
exception OBJ_ADAPTER ex_body;
                             // failure detected by object adapter
exception DATA_CONVERSION ex_body;
                             // data conversion error
exception OBJECT_NOT_EXIST ex_body;
                             // non-existent object, delete reference
exception TRANSACTION_REQUIRED ex_body;
                             // transaction required
exception TRANSACTION_ROLLEDBACK x_body;
                             // transaction rolled back
exception INVALID_TRANSACTION ex_body;
                             // invalid transaction
exception INV_POLICY ex_body;
```

```
                                        // invalid policy
    exception CODESET_INCOMPATIBLE ex_body
                                           // incompatible code set
    exception REBIND ex_body;
                                           // rebind needed
    exception TIMEOUT ex_body;
                                           // operation timed out
    exception TRANSACTION_UNAVAILABLE ex_body;
                                           // no transaction
    exception TRANSACTION_MODE ex_body;
                                           // invalid transaction mode
    exception BAD_QOS ex_body;
                                           // bad quality of service
    exception INVALID_ACTIVITY ex_body;
                                           // bad quality of service
    exception ACTIVITY_COMPLETED ex_body;
                                           // bad quality of service
    exception ACTIVITY_REQUIRED ex_body;
                                           // bad quality of service
};
```

8.6.3.1 UNKNOWN

This exception is raised if an operation implementation throws a non-CORBA exception (such as an exception specific to the implementation's programming language), or if an operation raises a user exception that does not appear in the operation’s raises expression. UNKNOWN is also raised if the server returns a system exception that is unknown to the client. (This can happen if the server uses a later version of CORBA than the client and new system exceptions have been added to the later version.)

8.6.3.2 BAD_PARAM

A parameter passed to a call is out of range or otherwise considered illegal. An ORB may raise this exception if null values or null pointers are passed to an operation (for language mappings where the concept of a null pointers or null values applies). BAD_PARAM can also be raised as a result of client generating requests with incorrect parameters using the DII.

8.6.3.3 NO_MEMORY

The ORB run time has run out of memory.

8.6.3.4 IMP_LIMIT

This exception indicates that an implementation limit was exceeded in the ORB run time. For example, an ORB may reach the maximum number of references it can hold simultaneously in an address space, the size of a parameter may have exceeded the allowed maximum, or an ORB may impose a maximum on the number of clients or servers that can run simultaneously.

8.6.3.5 COMM_FAILURE

This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.

8.6.3.6 INV_OBJREF

This exception indicates that an object reference is internally malformed. For example, the repository ID may have incorrect syntax or the addressing information may be invalid.

An ORB may choose to detect calls via nil references (but is not obliged to detect them). INV_OBJREF is used to indicate this.

If the client invokes an operation that results in an attempt by the client ORB to marshal wchar or wstring data for an in parameter (or to unmarshal wchar or wstring data for an in/out parameter, out parameter, or the return value), and the associated object reference does not contain a codeset component, the INV_OBJREF standard system exception is raised.

8.6.3.7 NO_PERMISSION

An invocation failed because the caller has insufficient privileges.

8.6.3.8 INTERNAL

This exception indicates an internal failure in an ORB, for example, if an ORB has detected corruption of its internal data structures.

8.6.3.9 MARSHAL

A request or reply from the network is structurally invalid. This error typically indicates a bug in either the client-side or server-side run time. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception. MARSHAL can also be caused by using the DII or DSI incorrectly, for example, if the type of the actual parameters sent does not agree with IDL signature of an operation.

8.6.3.10 INITIALIZE

An ORB has encountered a failure during its initialization, such as failure to acquire networking resources or detecting a configuration error.

8.6.3.11 NO_IMPLEMENT

This exception indicates that even though the operation that was invoked exists (it has an IDL definition), no implementation for that operation exists. NO_IMPLEMENT can, for example, be raised by an ORB if a client asks for an object's type definition from the interface repository, but no interface repository is provided by the ORB.

8.6.3.12 BAD_TYPECODE

The ORB has encountered a malformed type code (for example, a type code with an invalid **TCKind** value).

8.6.3.13 BAD_OPERATION

This indicates that an object reference denotes an existing object, but that the object does not support the operation that was invoked.

8.6.3.14 NO_RESOURCES

The ORB has encountered some general resource limitation. For example, the run time may have reached the maximum permissible number of open connections.

8.6.3.15 NO_RESPONSE

This exception is raised if a client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available.

8.6.3.16 PERSIST_STORE

This exception indicates a persistent storage failure, for example, failure to establish a database connection or corruption of a database.

8.6.3.17 BAD_INV_ORDER

This exception indicates that the caller has invoked operations in the wrong order. For example, it can be raised by an ORB if an application makes an ORB-related call without having correctly initialized the ORB first.

8.6.3.18 TRANSIENT

TRANSIENT indicates that the ORB attempted to reach an object and failed. It is not an indication that an object does not exist. Instead, it simply means that no further determination of an object's status was possible because it could not be reached. This exception is raised if an attempt to establish a connection fails, for example, because the server or the implementation repository is down.

8.6.3.19 FREE_MEM

The ORB failed in an attempt to free dynamic memory, for example because of heap corruption or memory segments being locked.

8.6.3.20 INV_IDENT

This exception indicates that an IDL identifier is syntactically invalid. It may be raised if, for example, an identifier passed to the interface repository does not conform to IDL identifier syntax, or if an illegal operation name is used with the DII.

8.6.3.21 INV_FLAG

An invalid flag was passed to an operation (for example, when creating a DII request).

8.6.3.22 INTF_REPOS

An ORB raises this exception if it cannot reach the interface repository, or some other failure relating to the interface repository is detected.

8.6.3.23 BAD_CONTEXT

An operation may raise this exception if a client invokes the operation but the passed context does not contain the context values required by the operation.

8.6.3.24 OBJ_ADAPTER

This exception typically indicates an administrative mismatch. For example, a server may have made an attempt to register itself with an implementation repository under a name that is already in use, or is unknown to the repository. OBJ_ADAPTER is also raised by the POA to indicate problems with application-supplied servant managers.

8.6.3.25 DATA_CONVERSION

This exception is raised if an ORB cannot convert the representation of data as marshaled into its native representation or vice-versa. For example, DATA_CONVERSION can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.

8.6.3.26 OBJECT_NOT_EXIST

The OBJECT_NOT_EXIST exception is raised whenever an invocation on a deleted object was performed. It is an authoritative “hard” fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate “final recovery” style procedures.

Bridges forward this exception to clients, also destroying any records they may hold (for example, proxy objects used in reference translation). The clients could in turn purge any of their own data structures.

8.6.3.27 TRANSACTION_REQUIRED

The TRANSACTION_REQUIRED exception indicates that the request carried a null transaction context, but an active transaction is required.

8.6.3.28 TRANSACTION_ROLLEDBACK

The TRANSACTION_ROLLEDBACK exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

8.6.3.29 INVALID_TRANSACTION

The INVALID_TRANSACTION indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

8.6.3.30 INV_POLICY

INV_POLICY is raised when an invocation cannot be made due to an incompatibility between Policy overrides that apply to the particular invocation.

8.6.3.31 CODESET_INCOMPATIBLE

This exception is raised whenever meaningful communication is not possible between client and server native code sets. See Section 13.7.2.6, “Code Set Negotiation,” on page 13-34.

8.6.3.32 REBIND

REBIND is raised when there is a problem in carrying out a requested or implied attempt to rebind an object reference (Section 7.4.1.2, “interface RebindPolicy,” on page 13).

8.6.3.33 TIMEOUT

TIMEOUT is raised when no delivery has been made and the specified time-to-live period has been exceeded. It is a standard system exception because time-to-live QoS can be applied to any invocation.

8.6.3.34 TRANSACTION_UNAVAILABLE

TRANSACTION_UNAVAILABLE exception is raised by the ORB when it cannot process a transaction service context because its connection to the Transaction Service has been abnormally terminated.

8.6.3.35 TRANSACTION_MODE

TRANSACTION_MODE exception is raised by the ORB when it detects a mismatch between the **TransactionPolicy** in the IOR and the current transaction mode.

8.6.3.36 BAD_QOS

The BAD_QOS exception is raised whenever an object cannot support the quality of service required by an invocation parameter that has a quality of service semantics associated with it.

8.6.3.37 INVALID_ACTIVITY

The INVALID_ACTIVITY system exception may be raised on the Activity or Transaction services’ resume methods if a transaction or Activity is resumed in a context different to that from which it was suspended. It is also raised when an attempted invocation is made that is incompatible with the Activity’s current state.

8.6.3.38 ACTIVITY_COMPLETED

The ACTIVITY_COMPLETED system exception may be raised on any method for which Activity context is accessed. It indicates that the Activity context in which the method call was made has been completed due to a timeout of either the Activity itself or a transaction that encompasses the Activity, or that the Activity completed in a manner other than that originally requested.

8.6.3.39 ACTIVITY_REQUIRED

The ACTIVITY_REQUIRED system exception may be raised on any method for which an Activity context is required. It indicates that an Activity context was necessary to perform the invoked operation, but one was not found associated with the calling thread.

8.6.4 Standard Minor Exception Codes

Please refer to Annex A for a table that specifies standard minor exception codes that have been assigned for the standard system exceptions.

8.7 Consolidated IDL

```
module CORBA {

#if ! defined(CORBA_E_MICRO)
    interface TypeCode;          // forward declaration
#endif

    typedef short PolicyErrorCode;

    typedef unsigned long PolicyType;

    exception PolicyError {PolicyErrorCode reason;};

    typedef string RepositoryId;
    typedef string Identifier;

    typedef unsigned short ServiceType;
    typedef unsigned long ServiceOption;
    typedef unsigned long ServiceDetailType;
    typedef CORBA::OctetSeq ServiceDetailData;
    typedef sequence<ServiceOption> ServiceOptionSeq;

    struct ServiceDetail {
        ServiceDetailType service_detail_type;
        ServiceDetailData service_detail;
    };

    typedef sequence<ServiceDetail> ServiceDetailSeq;

    struct ServiceInformation {
        ServiceOptionSeq service_options;
        ServiceDetailSeq service_details;
    };

#if ! defined(CORBA_E_MICRO)
    native ValueFactory;
#endif
```

```
    typedef string ORBid;

    interface ORB {

        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        ORBid id();

        string object_to_string (
            in Object           obj
        );

        Object string_to_object (
            in string           str
        );

        // Service information operations

        boolean get_service_information (
            in ServiceType service_type,
            out ServiceInformation service_information
        );

        ObjectIdList list_initial_services ();

        // Initial reference operation

        Object resolve_initial_references (
            in ObjectId identifier
        ) raises (InvalidName);

        // Thread related operations

        boolean work_pending( );

        void perform_work();

        void run();

        void shutdown(
            in boolean          wait_for_completion
        );

        void destroy();

        // Policy related operations

#if ! defined(CORBA_E_MICRO)
```

```
        Policy create_policy(
            in PolicyType       type,
            in any              val
        ) raises (PolicyError);
#endif

#if ! defined(CORBA_E_MICRO)
        // Value factory operations

        ValueFactory register_value_factory(
            in RepositoryId id,
            in ValueFactory_factory
        );

        void unregister_value_factory(in RepositoryId id);

        ValueFactory lookup_value_factory(in RepositoryId id);
#endif

        void register_initial_reference(
            in ObjectId id,
            in Object obj
        ) raises (InvalidName);
    };

    enum TCKind {
        tk_null, tk_void,
        tk_short, tk_long, tk_ushort, tk_ulong,
        tk_float, tk_double, tk_boolean, tk_char,
        tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
        tk_struct, tk_union, tk_enum, tk_string,
        tk_sequence, tk_array, tk_alias, tk_except,
        tk_longlong, tk_ulonglong, tk_longdouble,
        tk_wchar, tk_wstring, tk_fixed,
        tk_value, tk_value_box,
        tk_native,
        tk_abstract_interface,
        tk_local_interface
        tk_component, tk_home,
        tk_event
    };

    typedef short ValueModifier;
        const ValueModifier VM_NONE = 0;
        const ValueModifier VM_CUSTOM = 1;
        const ValueModifier VM_ABSTRACT = 2;
        const ValueModifier VM_TRUNCATABLE = 3;

#if ! defined(CORBA_E_MICRO)
    interface TypeCode {
        exception       Bounds {};
        exception        BadKind {};

        // for all TypeCode kinds
        boolean equal (in TypeCode tc);

        boolean equivalent(in TypeCode tc);
        TypeCode get_compact_typecode();

        TCKind kind ();

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // tk_local_interface, tk_except
        // tk_component, tk_home and tk_event
        RepositoryId id () raises (BadKind);

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // tk_local_interface, tk_except
        // tk_component, tk_home and tk_event
        Identifier name () raises (BadKind);

        // for tk_struct, tk_union, tk_enum, tk_value,
        // tk_except and tk_event
        unsigned long member_count () raises (BadKind);
        Identifier member_name (in unsigned long index)
            raises(BadKind, Bounds);

        // for tk_struct, tk_union, tk_value,
        // tk_except and tk_event
        TypeCode member_type (in unsigned long index)
            raises (BadKind, Bounds);

        // for tk_union
        any member_label (in unsigned long index)
            raises(BadKind, Bounds);
        TypeCode discriminator_type () raises (BadKind);
        long default_index () raises (BadKind);

        // for tk_string, tk_wstring, tk_sequence, and tk_array
        unsigned long length () raises (BadKind);

        // for tk_sequence, tk_array, tk_value_box and tk_alias
        TypeCode content_type () raises (BadKind);

        // for tk_fixed
        unsigned short fixed_digits() raises(BadKind);
        short fixed_scale() raises(BadKind);

        // for tk_value and tk_event
        Visibility member_visibility(in unsigned long index)
```

```
            raises(BadKind, Bounds);
        ValueModifier type_modifier() raises(BadKind);
        TypeCode concrete_base_type() raises(BadKind);
    };
#endif
    const unsigned long OMGVMCID = 0x4f4d0000;

#define ex_body {unsigned long minor; completion_status completed;}

    enum completion_status {
        COMPLETED_YES,
        COMPLETED_NO,
        COMPLETED_MAYBE
    };

    enum exception_type {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION
    };
};

    exception UNKNOWN ex_body;               // the unknown exception
    exception BAD_PARAM ex_body;             // an invalid parameter was passed
    exception NO_MEMORY ex_body;             // dynamic memory allocation failure
    exception IMP_LIMIT ex_body;             // violated implementation limit
    exception COMM_FAILURE ex_body;          // communication failure
    exception INV_OBJREF ex_body;            // invalid object reference
    exception NO_PERMISSION ex_body;         // no permission for attempted op.
    exception INTERNAL ex_body;              // ORB internal error
    exception MARSHAL ex_body;               // error marshaling param/result
    exception INITIALIZE ex_body;            // ORB initialization failure
    exception NO_IMPLEMENT ex_body;          // operation implementation unavailable
    exception BAD_TYPECODE ex_body;          // bad typecode
    exception BAD_OPERATION ex_body;         // invalid operation
    exception NO_RESOURCES ex_body;          // insufficient resources for req.
    exception NO_RESPONSE ex_body;           // response to req. not yet available
    exception PERSIST_STORE ex_body;         // persistent storage failure
    exception BAD_INV_ORDER ex_body;         // routine invocations out of order
    exception TRANSIENT ex_body;             // transient failure - reissue request
    exception FREE_MEM ex_body;              // cannot free memory
    exception INV_IDENT ex_body;             // invalid identifier syntax
    exception INV_FLAG ex_body;              // invalid flag was specified
    exception INTF_REPOS ex_body;            // error accessing interface repository
    exception BAD_CONTEXT ex_body;           // error processing context object
    exception OBJ_ADAPTER ex_body;           // failure detected by object adapter
    exception DATA_CONVERSION ex_body;       // data conversion error
    exception OBJECT_NOT_EXIST ex_body;      // non-existent object, delete reference
    exception TRANSACTION_REQUIRED ex_body;  // transaction required
    exception TRANSACTION_ROLLEDBACK x_body; // transaction rolled back
    exception INVALID_TRANSACTION ex_body;   // invalid transaction
    exception INV_POLICY ex_body;                      // invalid policy
    exception CODESET_INCOMPATIBLE ex_body             // incompatible code set
    exception REBIND ex_body;                          // rebind needed
    exception TIMEOUT ex_body;                         // operation timed out
    exception TRANSACTION_UNAVAILABLE ex_body;  // no transaction
    exception TRANSACTION_MODE ex_body;                // invalid transaction mode
    exception BAD_QOS ex_body;                         // bad quality of service
    exception INVALID_ACTIVITY ex_body;                // bad quality of service
    exception ACTIVITY_COMPLETED ex_body;              // bad quality of service
    exception ACTIVITY_REQUIRED ex_body;               // bad quality of service
};
```

9 Object Interfaces

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter except 9.3, ’Value Type Semantics’.

9.1 Overview

This chapter introduces the operations and other facilities provided for the object-oriented constructs defined in OMG IDL and described in the Object Model. These operations are implemented by the ORB. In particular, it describes the base operations available for every instance of an interface and every instance of a value type.

9.2 Object (Reference) Interface

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface **Object** to represent the object reference, we define an interface for **Object**:

```
module CORBA {

    interface Policy;                       // forward declaration
    typedef sequence <Policy> PolicyList;
    typedef sequence<PolicyType> PolicyTypeSeq;
    exception InvalidPolicies { sequence <unsigned short> indices; };

    typedef string Identifier;
    typedef unsigned long Flags;

    enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

    interface ORB;        // PIDL forward declaration

    interface Object {                                          // PIDL

        InterfaceDef get_interface ();

        boolean is_nil();

        Object duplicate ();

        void release ();

        boolean is_a (
            in RepositoryId         logical_type_id
        );
        boolean non_existent();

        boolean is_equivalent (
            in Object               other_object
        );

        unsigned long hash(
            in unsigned long        maximum
        );

        Policy get_policy (
            in PolicyType           policy_type
        );

        Object set_policy_overrides(
            in PolicyList           policies,
            in SetOverrideType      set_add
        ) raises (InvalidPolicies);

        Policy get_client_policy(
            in PolicyType type
        );

        PolicyList get_policy_overrides(
            in PolicyTypeSeq        types
        );

        boolean validate_connection(
            out PolicyList           inconsistent_policies
        );

        string respository_id();
        ORB get_orb();
    };
};
```

Unless otherwise stated below, the operations in the IDL above do not require access to remote information.

9.2.1 Determining the Object Interface

9.2.1.1 get_interface

```
    InterfaceDef get_interface();
```

get_interface, returns an object in the Interface Repository that describes the most derived type of the object addressed by the reference. See the Interface Repository chapter for a definition of operations on the Interface Repository. The implementation of this operation may involve contacting the ORB that implements the target object.

If the interface repository is not available, **get_interface** raises INTF_REPOS with standard minor code 1. If the interface repository does not contain an entry for the object's (most derived) interface, **get_interface** raises INTF_REPOS with standard minor code 2.

9.2.1.2 repository_id

repository_id returns the repository ID of an object. The implementation of this operation must contact the ORB that implements the target object.

9.2.2 Duplicating and Releasing Copies of Object References

9.2.2.1 duplicate

```
Object duplicate();
```

9.2.2.2 release

```
void release();
```

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

If more than one copy of an object reference is needed, the client may create a **duplicate**. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

9.2.3 Nil Object References

9.2.3.1 is_nil

```
boolean is_nil();
```

An object reference whose value is **OBJECT_NIL** denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

9.2.4 Equivalence Checking Operation

9.2.4.1 is_a

```
boolean is_a(
    in RepositoryId         logical_type_id
);
```

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

The **logical_type_id** is a string denoting a shared type identifier (**RepositoryId**). The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the “most derived” type of that object.

Determining whether an object’s type is compatible with the **logical_type_id** may require contacting a remote ORB or interface repository. Such an attempt may fail at either the local or the remote end. If **is_a** cannot make a reliable determination of type compatibility due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the **TRUE**, **FALSE**, and indeterminate cases.

This operation exposes to application programmers functionality that must already exist in ORBs that support “type safe narrow” and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

This operation always returns **TRUE** for the **logical_type_id IDL:omg.org/CORBA/Object:1.0**

9.2.5 Probing for Object Non-Existence

9.2.5.1 non_existent

```
boolean non_existent ();
```

The **non_existent** operation may be used to test whether an object (e.g., a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising CORBA::OBJECT_NOT_EXIST) if the ORB knows authoritatively that the object does not exist; otherwise, it returns false.

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their “idle time” to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

Probing for object non-existence may require contacting the ORB that implements the target object. Such an attempt may fail at either the local or the remote end. If non-existent cannot make a reliable determination of object existence due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the true, false, and indeterminate cases.

9.2.6 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

9.2.6.1 Hashing Object Identifiers

hash

```
unsigned long hash(
    in unsigned long        maximum
);
```

Object references are associated with ORB-internal identifiers that may indirectly be accessed by applications using the **hash** operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are identical.

The **maximum** parameter to the **hash** operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given "real" object. Those proxies would not necessarily hash to the same value.

9.2.6.2 Equivalence Testing

is_equivalent

```
boolean is_equivalent(
    in Object                   other_object
);
```

The **is_equivalent** operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns **TRUE** if the target object reference is known to be equivalent to the other object reference passed as its parameter, and **FALSE** otherwise.

If two object references are identical, they are equivalent. Two different object references that in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. This means that a **FALSE** return from **is_equivalent** should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects. Setting of local policies on the object reference is not taken into consideration for the purposes of determining object reference equivalence.

A typical application use of this operation is to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to "flatten" graphs that represent cyclical relationships between objects. Some might do this as they construct the table, others during idle time.

9.2.7 Type Coercion Considerations

Many programming languages map **Object** to programming constructs that support inheritance. Mappings to languages (such as C++ and Java) typically provide a mechanism for narrowing (down-casting) an object reference from a base interface to a more derived interface. To do such down-casting in a type safe way, knowledge of the full inheritance hierarchy of the target interface may be required. The implementation of down-cast must either contact an interface repository or the target itself, to determine whether or not it is safe to down-cast the client's object reference. This requirement is not acceptable when a client is expecting only asynchronous communication with the target. Therefore, for the appropriate languages an unchecked down-cast operation (also referred to as unchecked narrow operation) shall be provided in the mapping of Object. This unchecked narrow always returns a stub of the requested type without checking that the target really implements that interface.

9.2.8 Getting Policy Associated with the Object

9.2.8.1 get_policy

The **get_policy** operation returns the policy object of the specified type (see Section 7.2, “Policy Object,” on page 5), which applies to this object. It returns the effective **Policy** for the object reference. The effective **Policy** is the one that would be used if a request were made.

This **Policy** is determined first by obtaining the effective override for the **PolicyType** as returned by **get_client_policy**. The effective override is then compared with the **Policy** as specified in the **IOR**. The effective **Policy** is determined by reconciling the effective override and the **IOR**-specified **Policy** (see Section 7.3.2, “Server Side Policy Management,” on page 9). If the two policies cannot be reconciled, the standard system exception INV_POLICY is raised with standard minor code 1. The absence of a **Policy** value in the **IOR** implies that any legal value may be used.

Invoking **non_existent** on an object reference prior to **get_policy** ensures the accuracy of the returned effective **Policy**. If **get_policy** is invoked prior to the object reference being bound, a compliant implementation shall attempt a binding and then return the effective **Policy**. If the binding attempt fails it shall pass through the system exception returned from the binding attempt. Note that if the effective **Policy** may change from invocation to invocation due to transparent rebinding.

```
Policy get_policy (
    in PolicyType        policy_type
);
```

Parameter(s)

policy_type - The type of policy to be obtained.

Return Value

A **Policy** object of the type specified by the **policy_type** parameter.

Exception(s)

CORBA::INV_POLICY - raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

The implementation of this operation may involve remote invocation of an operation (e.g., **DomainManager::get_domain_policy** for some security policies) for some policy types.

9.2.8.2 get_client_policy

```
Policy get_client_policy(
    in PolicyType type
);
```

Returns the **Policy** for the object reference. The effective override is obtained by first checking for an override of the given **PolicyType** at the **Object** scope, then at the **Current** scope, and finally at the ORB scope. If no override is present for the requested **PolicyType**, a system-dependent default value for that **Policy Typ**e may be returned. A nil **Policy** reference may also be returned to indicate that there is no default for the policy. Portable applications are expected to set the desired "defaults" at the ORB scope since default **Policy** values are not specified.

9.2.8.3 get_policy_overrides

```
PolicyList get_policy_overrides(
    in PolicyTypeSeq        types
);
```

Returns the list of **Policy** overrides (of the specified policy types) set at the **Object** scope. If the specified sequence is empty, all **Policy** overrides at this scope will be returned. If none of the requested **PolicyTypes** are overridden at the **Object** scope, an empty sequence is returned.

9.2.9 Overriding Associated Policies on an Object Reference

9.2.9.1 set_policy_overrides

The **set_policy_overrides** operation returns a new object reference with the new policies associated with it. It takes two input parameters. The first parameter **policies** is a sequence of references to **Policy** objects. The second parameter **set_add** of type **SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**ADD_OVERRIDE**) in the object reference, or they should be added to a clean override free object reference (**SET_OVERRIDE**). This operation associates the policies passed in the first parameter with a newly created object reference that it returns. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

```
enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

Object set_policy_overrides(
    in PolicyList               policies,
    in SetOverrideType          set_add
                                ) raises (InvalidPolicies);
```

Parameter(s)

policies - a sequence of **Policy** objects that are to be associated with the new copy of the object reference returned by this operation. If the sequence contains two or more **Policy** objects with the same **PolicyType** value, the operation raises the standard system exception BAD_PARAM with minor code 30.

set_add - whether the association is in addition to (**ADD_OVERRIDE**) or as a replacement of (**SET_OVERRIDE**) any existing overrides already associated with the object reference. If the value of this parameter is **SET_OVERRIDE**, the supplied **policies** completely replace all existing overrides associated with the object reference. If the value of this parameter is **ADD_OVERRIDE**, the supplied **policies** are added to the existing overrides associated with the object reference, except that if a supplied **Policy** object has the same **PolicyType** value as an existing override, the supplied **Policy** object replaces the existing override.

Return Value

A copy of the object reference with the overrides from **policies** associated with it in accordance with the value of **set_add**.

Exception(s)

InvalidPolicies - raised when an attempt is made to override any policy that cannot be overridden.

9.2.10 Validating Connection

9.2.10.1 validate_connection

```
boolean validate_connection(
    out PolicyList          inconsistent_policies
);
```

Returns the value TRUE if the current effective policies for the **Object** will allow an invocation to be made. If the object reference is not yet bound, a binding will occur as part of this operation. If the object reference is already bound, but current policy overrides have changed or for any other reason the binding is no longer valid, a rebind will be attempted regardless of the setting of any **RebindPolicy** override. The **validate_connection** operation is the only way to force such a rebind when implicit rebinds are disallowed by the current effective **RebindPolicy**. The attempt to bind or rebind may involve processing GIOP LocateRequests by the ORB.

If the RoutingPolicy **ROUTE_FORWARD** or **ROUTE_STORE_AND_FORWARD** are in effect when **validate_connection** is invoked, then the client ORB shall attempt to open a connection for the first hop to the first target **Router** (applies to both **Router** and **PersistentRequestRouter**) as if it were the target **Object** and return success or failure based on success or failure to establish this connection.

Returns the value FALSE if the current effective policies would cause an invocation to raise the standard system exception INV_POLICY. If the current effective policies are incompatible, the out parameter **inconsistent_policies** contains those policies causing the incompatibility. This returned list of policies is not guaranteed to be exhaustive. If the binding fails due to some reason unrelated to policy overrides, the appropriate standard system exception is raised.

9.2.11 Getting the ORB

9.2.11.1 get_orb

```
ORB get_orb();
```

This operation returns the local ORB that is handling this particular Object Reference.

9.2.12 LocalObject Operations

Local interfaces are implemented by using **CORBA::LocalObject**, which derives from **CORBA::Object** and provides implementations of Object pseudo operations and any other ORB specific support mechanisms that are appropriate for such objects. Object implementation techniques are inherently language mapping specific. Therefore, the **LocalObject** type is not defined in IDL, but is specified by each language mapping.

- The **LocalObject** type provides implementations of the following **Object** pseudo-operations that raise the NO_IMPLEMENT system exception with standard minor code 8:
    - **get_interface**
    - **get_domain_managers**
    - **get_policy**
    - **get_client_policy**
    - **set_policy_overrides**
    - **get_policy_overrides**
    - **validate_connection**
    - **get_component**
    - **respository_id**
- The **LocalObject** type provides implementations of the following pseudo-operations:
    - **non_existent** - always returns false.
    - **hash** - returns a hash value that is consistent for the lifetime of the object.
    - **is_equivalent** - returns true if the references refer to the same **LocalObject** implementation.
- **is_a** - returns **TRUE** if the **LocalObject** derives from or is itself the type specified by the **logical_type_id** argument.
- **get_orb** - The default behavior of this operation when invoked on a reference to a local object is to return the system exception NO_IMPLEMENT with standard minor code 8. Certain local objects that have close association with an ORB, like POAs, Current objects and certain portable interceptors related local objects override the default behavior and return a reference to the ORB that they are associated with. These are documented in the sections where these local objects are specified
- Attempting to use a LocalObject to create a DII request shall result in a NO_IMPLEMENT system exception with standard minor code 4. Attempting to marshal or stringify a LocalObject shall result in a MARSHAL system exception with standard minor code 4. Narrowing and widening of references to **LocalObject**s must work as for regular object references.
- Local types cannot be marshaled and references to local objects cannot be converted to strings. Any attempt to marshal a local object, such as via an unconstrained base interface, as an **Object**, or as the contents of an **any**, or to pass a local object to **ORB::object_to_string**, shall result in a MARSHAL system exception with OMG minor code 4.
- The DII is not supported on local objects, nor are asynchronous invocation interfaces.
- Language mappings shall specify server side mechanisms, including base classes and/or skeletons if necessary, for implementing local objects, so that invocation overhead is minimized.
- The usage of client side language mappings for local types shall be identical to those of equivalent unconstrained types.
- Invocations on local objects are not ORB mediated. Specifically, parameter copy semantics are not honored, interceptors are not invoked, and the execution context of a local object does not have ORB service **Current** object contexts that are distinct from those of the caller. Implementations of local interfaces are responsible for providing the parameter copy semantics expected by clients.
- Local objects have no inherent identities beyond their implementations' identities as programming objects. The lifecycle of the implementation is the same as the lifecycle of the reference.

- Instances of local objects defined as part of OMG specifications to be supplied by ORB products or object service products shall be exposed through the **ORB::resolve_initial_references** operation or through some other local object obtained from **resolve_initial_references**.

9.3 Value Type Semantics

9.3.1 Overview

Objects, more specifically, interface types that objects support, are defined by an IDL interface, allowing arbitrary implementations. There is great value, which is described in great detail elsewhere, in having a distributed object system that places almost no constraints on implementations.

However there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object's primary "purpose" is to encapsulate data, or an application explicitly wishes to make a "copy" of an object.

The semantics of passing an object by value are similar to that of standard programming languages. The receiving side of a parameter passed by value receives a description of the "state" of the object. It then instantiates a new instance with that state but having a separate identity from that of the sending side. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances.

Because it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object's state and implementation.

Value types provide semantics that bridge between CORBA structs and CORBA interfaces:

- They support description of structured state
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call)
- They support both public and private (to the implementation) data members.
- They support single inheritance (of **valuetype**)
- They may also be **abstract**.

The **ValueBase** interface contains operations that are implemented by the ORB and are accessed as implicit operations of the **ValueBase** Reference.

9.3.2 Architecture

The basic notion is relatively simple. A **value type** is, in some sense, half way between a "regular" IDL interface type and a struct. The use of a value type is a signal from the designer that some additional properties (state) and implementation details be specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

An essential property of value types is that their implementations are always local. That is, the explicit use of value type in a concrete programming language is always guaranteed to use a local implementation, and will not require a remote call. They have no identity (their value is their identity) and they are not "registered" with the ORB.

There are two kinds of value types, concrete (or stateful) value types, and abstract (stateless) ones. As explained below the essential characteristics of both are the same. The differences between them result from the differences in the way they are mapped in the language mappings. In this specification the semantics of value types apply to both kinds, unless specifically stated otherwise.

Concrete (stateful) values add to the expressive power of (IDL) structs by supporting:

- single derivation (from other value types)
- null value semantics

When an instance of such a type is passed as a parameter, the sending context marshals the state (data) and passes it to the receiving context. The receiving context instantiates a new instance using the information in the GIOP request and unmarshals the state. It is assumed that the receiving context has available to it an implementation that is consistent with the sender's (i.e., only needs the state information), or that it can somehow download a usable implementation.

It should be noted that it is possible to define a concrete value type with an empty state as a degenerate case.

9.3.2.1 Abstract Values

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. Only concrete types derived from them may be actually instantiated and implemented. Their implementation, of course, is still local. However, because no state information may be specified (only local operations are allowed), abstract value types are not subject to the single inheritance restrictions placed upon concrete value types. Essentially they are a bundle of operation signatures with a purely local implementation. This distinction is made clear in the language mappings for abstract values.

Note that a concrete value type with an empty state is not an abstract value type. They are considered to be stateful, may be instantiated, marshaled, and passed as actual parameters. Consider them to be a degenerate case of stateful values.

9.3.2.2 Operations

Operations defined on a value type specify signatures whose implementation can only be local. Because these operations are local, they must be directly implemented by a body of code in the language mapping (no proxy or indirection is involved).

The language mappings of such operations require that instances of value types passed into and returned by such local methods are passed by reference (programming language reference semantics, not CORBA object reference semantics) and that a copy is not made. Note, such a (local) invocation is not a CORBA invocation. Hence it is not mediated by the ORB, although the API to be used is specified in the language mapping.

The (copy) semantics for instances of value type are only guaranteed when instances of these value types are passed as a parameter to an operation defined on a CORBA interface, and hence mediated by the ORB. If an instance of a value type is passed as a parameter to a method of another value type in an invocation, then this call is a “normal” programming language call. In this case both of the instances are local programming language constructs. No CORBA style copy semantics are used and programming language reference semantics apply.

Operations on the value type are supported in order to guarantee the portability of the client code for these value types. They have no representation on the wire and hence no impact on interoperability.

9.3.2.3 Value Type vs. Interfaces

By default value types are not CORBA Objects. In particular instances of value types do not inherit from **CORBA::Object** and do not support normal object reference semantics. CORBA/*e* does not support the “supports” clause for derivation of a valuetype from an interface.

9.3.2.4 Parameter Passing

This section describes semantics when a value instance is passed as parameter in a CORBA invocation. It does not deal with the case of calling another non-CORBA (i.e., local) programming method, which happens to have a parameter of the same type.

Value vs. Reference Semantics

Determination of whether a parameter is to be passed by value or reference is made by examining the parameter’s formal type (i.e., the signature of the operation it is being passed to). If it is a value type, then it is passed by value. If it is an ordinary interface, then it is passed by reference (the case today for all CORBA objects). This rule is simple and consistent with the handling of the same situation in recursive state definitions or in structs.

Sharing Semantics

In order to be amenable to implementation on resource constrained systems, value types in the CORBA/*e* profiles **do not** support sharing (see 9.3.2.5, ’Value Types as Members of Other Types’). They do support null semantics; i.e., an instance can be null.

Identity Semantics

When an instance of the value type is passed as a parameter to an operation of a non-local interface, the effect in all cases shall be as if an independent copy of the instance is instantiated in the receiving context. While certain implementation optimizations are possible the net effect shall be as if the copy is a separate independent entity and there is no explicit or implicit sharing of state. This applies to all valuetypes involved in the invocation, including those embedded in other IDL datatypes or in an any. This notional copying occurs twice, once for in and inout parameters when the invocation is initiated, and once again for inout, out, and return parameters when the invocation completes. Optimization techniques such as copy on write, etc. must make sure that the semantics of copying as described above is preserved.

Any parameter type

When an instance of a value type is passed to an **any**, as with all cases of passing instances to an **any**, it is the responsibility of the implementor to insert and extract the value according to the language mapping specification.

9.3.2.5 Value Types as Members of Other Types

Syntactically, instances of value types may be included as members of other value types or as members of any IDL constructed type, e.g., structs and unions. At run-time, this capability, combined with the sharing semantics supported by other CORBA profiles would allow the formation of polylithic instances of value types to represent trees, lists, and other graphs.

Support of instances of graphs with arbitrary cyclic topology would require implementation of complex memory management techniques, such as garbage collection. In order to alleviate this concern, the following constraint is placed on value type definitions:

> No set of IDL files shall introduce a member of a value type that contains an instance of any value type, an instance of type Any, or contains any other constructed type that, either as a member or transitively as a member of an included type, contains an instance of any value type or of type Any.

9.3.2.6 Substitutability Issues

The substitutability requirements for CORBA require the definition of what happens when an instance of a derived value type is passed as a parameter that is declared to be a base value type.

If the receiving context currently has the appropriate implementation class, then there is no problem.

If the receiving context does not currently hold an implementation with which to reconstruct the original type, then the following algorithm is used to find such an implementation:

1. ***Load*** - Attempt to load (locally in C/C++, possibly remotely in Java and other "portable" languages) the real type of the object (with its methods). If this succeeds, OK.
2. ***Truncate*** - Truncate the type of the object to the base type (if specified as **truncatable** in the IDL). Truncation can never lead to faulty programs because, from a structural point view base types structurally subsume a derived type and an object created in the receiving context bears no relationship with the original one. However, it might be semantically puzzling, as the derived type may completely re-interpret the meaning of the state of the base. For that reason a derived value needs to indicate if it is safe to truncate to its immediate non-abstract parent.
3. ***Raise Exception*** - If none of these work or are possible, then raise the NO_IMPLEMENT exception with standard minor code 1.

Truncatability is a transitive property.

Example

```
valuetype EmployeeRecord {   // note this is not a CORBA::Object
    // state definition
    private string name;
    private string email;
    private string SSN;
    // initializer
    factory init(in string name, in string SSN);
};

valuetype ManagerRecord: truncatable EmployeeRecord {
    // state definition
    private sequence<EmployeeRecord> direct_reports;
};
```

9.3.2.7 Widening/Narrowing

As has been described above, value type instances may be widened/narrowed to other value types. Each language mapping is responsible for specifying how these operations are made available to the programmer.

9.3.2.8 Value Base Type

All value types have a conventional base type called **ValueBase**. This is a type, which fulfills a role that is similar to that played by **Object**. Conceptually it supports the common operations available on all value types. In each language mapping **ValueBase** will be mapped to an appropriate base type that supports the marshaling/unmarshaling protocol.

The mapping for other operations, which all value types must support, such as getting meta information about the type, may be found in the specifics for each language mapping.

Typically it is mapped to a concrete language type which serves as a base for all value types. Any operations that are required to be supported for all values are conceptually defined on **ValueBase**, although in reality their actual mapping depends upon the specifics of any particular language mapping.

Analogous to the definition of the **Object** interface for implicit operations of object references, the implicit operations of **ValueBase** are defined on a pseudo-**valuetype** as follows:

```
module CORBA {
    valuetype ValueBase{                                    PIDL
        ValueDef get_value_def();
    };
};
```

The **get_value_def()** operation returns a description of the value's definition as described in the interface repository.

9.3.2.9 Life Cycle issues

Value type instances are always local to their creating context. For example, in a given language mapping an instance of a value type is always created as a local "language" object with no POA semantics attached to it initially.

When passed using a CORBA invocation, a copy of the value is made in the receiving context and that copy starts its life as a local programming language entity with no POA semantics attached to it.

Creation and Factories

When an instance of a value type is received by the ORB, it must be unmarshaled and an appropriate factory for its actual type found in order for the new instance to be created. The type is encoded by the RepositoryID, which is passed over the wire as part of an invocation. The mapping between the type (as specified by the RepositoryID) and the factory is language specific. In certain languages it may be possible to specify default policies that are used to find the factory, without requiring that specific routines be called. In others the runtime and/or generated code may have to explicitly specify the mapping on a per type basis. In others a combination may be used. In any event the ORB implementation is responsible for maintaining this mapping. See Section 9.3.3.3, "Language Specific Value Factory Requirements," on page 151 for more details on the requirements for each language mapping. Value box types do not need or use factories.

9.3.2.10 Security Considerations

The addition of value types has few impacts on the CORBA security model. In essence, the security implications in defining and using value types are similar to those involved with the use of IDL structs. Instances of value types are mapped to local, concrete programming language constructs. Except for providing the marshaling mechanisms, the ORB is not directly involved with accessing value type implementations. This specification is mostly about two things: how value types manifest themselves as concrete programming language constructs and how they are transmitted.

To see this consider how value types are actually used. The IDL definition of a value type in conjunction with a programming language mapping is used to generate the concrete programming language definitions for that type.

Let us consider its life cycle. In order to use it, the programmer uses the mechanisms in the programming language to instantiate an instance. This is instance is a local programming language construct. It is not “registered” with the ORB, object adapter, etc. The programmer may manipulate this programming construct just like any other programming language construct. So far there are no security implications. As long as no ORB-mediated invocations are made, the programmer may manipulate the construct. Note, this includes making “local,” non ORB-mediated calls to any locally implemented operations. Any assignments to the construct are the responsibility of the programmer and have no special security implications.

Things get interesting when the program attempts to pass one of these constructs through an orb-mediated invocation (i.e., calls a stub that uses it as a parameter type).

The formal type of the parameter is a value. This case is no different from using any other kind of a value (long, string, struct) in a CORBA invocation, with respect to security. The value (data) is marshaled and delivered to the receiving context. On the receiving context, the knowledge of the type is used (at least implicitly) to find the factory to create the correct local programming language construct. The data is then unmarshaled to fill in the newly created construct. This is similar to using other values (longs, strings, structs) except that the knowledge of the factory is not “built-in” to the ORB’s skeleton/DSI engine.

9.3.3 Language Mappings

9.3.3.1 General Requirements

A concrete value is mapped to a concrete usable “class” construct in each programming language, plus possibly some helper classes where appropriate. In Java, C++, and Smalltalk this is a real concrete class. In C it is a struct.

An abstract value is mapped to some sort of an abstract construct--an interface in Java, and an abstract class with pure virtual function members in C++.

Tools that implement the language mapping are free to “extend” the implementation classes with “extra” data members and methods. When an instance of such a class is used as a parameter, only the portions that correspond directly to the IDL declaration, are marshaled and delivered to the receiving context. This allows freedom of implementations while preserving the notion of contract and type safety in IDL.

9.3.3.2 Language Specific Marshaling

Each language mapping defines an appropriate marshaling/unmarshaling API and the entry point for custom marshaling/unmarshaling.

9.3.3.3 Language Specific Value Factory Requirements

Each language mapping specifies the algorithm and means by which RepositoryIDs are used to find the appropriate factory for an instance of a value type so that it may be created as it is unmarshaled “off the wire.”

It is desirable, where it makes sense, to specify a “default” policy for automatically using RepositoryIDs that are in common formats to find the appropriate factory. Such a policy can be thought of as an implicit registration.

Each language mapping specifies how and when the registration occurs, both explicit and implicit. The registration must occur before an attempt is made to unmarshal an instance of a value type. If the ORB is unable to locate and use the appropriate factory, then a MARSHAL exception with standard minor code 1 is raised.

Because the type of the factory is programming language specific and each programming language platform has different policies, the factory type is specified as **native**. It is the responsibility of each language mapping to specify the actual programming language type of the factory.

```
module CORBA {

    // IDL
    native ValueFactory;
};
```

9.3.3.4 Value Method Implementation

The mapped class must support method bodies (i.e., code) that implement the required IDL operations. The means by which this association is accomplished is a language mapping "detail" in much the same way that an IDL compiler is.

9.4 Consolidated IDL

```
module CORBA {

    interface Policy;                       // forward declaration
    typedef sequence <Policy> PolicyList;
    typedef sequence <PolicyType> PolicyTypeSeq;
    exception InvalidPolicies { sequence <unsigned short> indices; };

    typedef string Identifier;
    typedef unsigned long Flags;

    enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

    interface ORB;        // PIDL forward declaration

    interface Object {                                        // PIDL

        InterfaceDef get_interface ();

        boolean is_nil();

        Object duplicate ();

        void release ();

        boolean is_a (
            in RepositoryId          logical_type_id
        );
        boolean non_existent();

        boolean is_equivalent (
            in Object                   other_object
        );

        unsigned long hash(
            in unsigned long            maximum
        );

        Policy get_policy (
            in PolicyType               policy_type
        );

        Object set_policy_overrides(
            in PolicyList               policies,
            in SetOverrideType          set_add
        ) raises (InvalidPolicies);

        Policy get_client_policy(
            in PolicyType type
        );

        PolicyList get_policy_overrides(
            in PolicyTypeSeq            types
        );

        boolean validate_connection(
            out PolicyList               inconsistent_policies
        );

        string respository_id();
        ORB get_orb();
    };
};
```

10 Policies

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter.

Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter.

10.1 Overview

10.1.1 Usages of Policy Objects

Policy Objects are used in general to encapsulate information about a specific policy, with an interface derived from the policy interface. The type of the Policy object determines how the policy information contained within it is used. Usually a Policy object is associated with another object to associate the contained policy with that object.

Objects with which policy objects are typically associated are Domain Managers, POA, the execution environment, both the process/capsule/ORB instance and thread of execution (Current object) and object references. Only certain types of policy object can be meaningfully associated with each of these types of objects.

These relationships are documented in sections that pertain to these individual objects and their usages in various core facilities and object services. The use of Policy Objects with the POA are discussed in the *Portable Object Adapter* chapter. The use of Policy objects in the context of the Security services, involving their association with Domain Managers as well as with the Execution Environment are discussed in the *Security Service* specification.

In the following section the association of Policy objects with the Execution Environment is discussed. In Section 10.3, “Management of Policies,” on page 160 the use of Policy objects in association with Domain Managers is discussed.

10.1.2 Policy Associated with the Execution Environment

Certain policies that pertain to services like security (e.g., QOP, Mechanism, invocation credentials, etc.) are associated by default with the process/capsule(RM-ODP)/ORB instance (hereinafter referred to as “capsule”) when the application is instantiated together with the capsule. By default these policies are applicable whenever an invocation of an operation is attempted by any code executing in the said capsule. The Security service provides operations for modulating these policies on a per-execution thread basis using operations in the **Current** interface. Certain of these policies (e.g., invocation credentials, qop, mechanism, etc.) which pertain to the invocation of an operation through a specific object reference can be further modulated at the client end, using the **set_policy_overrides** operation of the **Object** reference. It associates a specified set of policies with a newly created object reference that it returns.

The association of these overridden policies with the object reference is a purely local phenomenon. These associations are never passed on in any IOR or any other marshaled form of the object reference. the associations last until the object reference in the capsule is destroyed or the capsule in which it exists is destroyed.

The policies thus overridden in this new object reference and all subsequent duplicates of this new object reference apply to all invocations that are done through these object references. The overridden policies apply even when the default policy associated with **Current** is changed. It is always possible that the effective policy on an object reference at any given time will fail to be successfully applied, in which case the invocation attempt using that object reference will fail and return a CORBA::NO_PERMISSION exception. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. These are listed in the Security specification. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

In general the policy of a specific type that will be used in an invocation through a specific object reference using a specific thread of execution is determined first by determining if that policy type has been overridden in that object reference. if so then the overridden policy is used. if not then if the policy has been set in the thread of execution then that policy is used. If not then the policy associated with the capsule is used. For policies that matter, the ORB ensures that there is a default policy object of each type that matters associated with each capsule (ORB instance). Hence, in a correctly implemented ORB there is no case when a required type policy is not available to use with an operation invocation.

10.1.3 Policy Associated with Message Interactions

This section describes a standard Quality of Service (QoS) framework within which CORBA Services specifications should define their service-specific qualities. In this framework, all QoS settings are interfaces derived from **CORBA::Policy**.

The details of the Policy Management Framework are to be found in the *ORB Interface* chapter.

Messaging requires clients and servers to have the ability to set the required and supported qualities of service with respect to requests. This specification provides generalized APIs through which such qualities are set in clients and servers. In addition, the set of Messaging-related qualities and the rules for reconciling and using these qualities are defined. Finally, the Messaging-specific IOR Profile Component and Service Context are defined for propagation of QoS information.

10.1.4 Specification of New Policy Objects

When new **PolicyType**s are added to CORBA specifications, the following details must be defined. It must be clearly stated which particular uses of a new policy are legal and which are not:

- Specify the assigned **CORBA::PolicyType** and the policy's interface definition.
- If the **Policy** can be created through **CORBA::ORB::create_policy**, specify the allowable values for the any argument 'val' and how they correspond to the initial state/behavior of that **Policy** (such as initial values of attributes). For example, if a Policy has multiple attributes and operations, it is most likely that create_policy will receive some complex data for the implementation to initialize the state of the specific policy:

```
//IDL
struct MyPolicyRange {
      long low;
      long high;
};

const CORBA::PolicyType MY_POLICY_TYPE = 666;
interface MyPolicy : Policy {
      readonly attribute long low;
      readonly attribute long high;
};
```

If this sample **MyPolicy** can be constructed via create_policy, the specification of **MyPolicy** will have a statement such as: "When instances of **MyPolicy** are created, a value of type **MyPolicyRang**e is passed to **CORBA::ORB::create_policy** and the resulting MyPolicy's attribute 'low' has the same value as the **MyPolicyRange** member 'low' and attribute 'high' has the same value as the **MyPolicyRange** member 'high.'

- If the **Policy** can be passed as an argument to **POA::create_POA**, specify the effects of the new policy on that **POA**. Specifically define incompatibilities (or inter-dependencies) with other **POA** policies, effects on the behavior of invocations on objects activated with the **POA**, and whether or not presence of the POA policy implies some **IOR** profile/component contents for object references created with that **POA**. If the **POA** policy implies some addition/ modification to the object reference it is marked as "client-exposed" and the exact details are specified including which profiles are affected and how the effects are represented.
- If the component that is used to carry this information can be set within a client to tune the client's behavior, specify the policy's effects on the client specifically with respect to (a) establishment of connections and reconnections for an object reference; (b) effects on marshaling of requests; (c) effects on insertion of service contexts into requests; (d) effects upon receipt of service contexts in replies. In addition, incompatibilities (or inter-dependencies) with other client-side policies are stated. For policies that cause service contexts to be added to requests, the exact details of this addition are given.
- If the **Policy** can be used with **POA** creation to tune **IOR** contents and can also be specified (overridden) in the client, specify how to reconcile the policy's presence from both the client and server. It is strongly recommended to avoid this case! As an exercise in completeness, most **POA** policies can probably be extended to have some meaning in the client and vice versa, but this does not help make usable systems, it just makes them more complicated without adding really useful features. There are very few cases where a policy is really appropriate to specify in both places, and for these policies the interaction between the two must be described.
- Pure client-side policies are assumed to be immutable. This allows efficient processing by the runtime that can avoid re-evaluating the policy upon every invocation and instead can perform updates only when new overrides are set (or policies change due to rebind). If the newly specified policy is mutable, it must be clearly stated what happens if non-readonly attributes are set or operations are invoked that have side-effects.
- For certain policy types, override operations may be disallowed. If this is the case, the policy specification must clearly state what happens if such overrides are attempted.

10.1.5 Standard Policies

Note – See Annex A for a list of the standard policy types that are defined by various parts of CORBA and CORBAservices in this version of CORBA.

10.2 Policy Object

10.2.1 Definition of Policy Object

An ORB or CORBA service may choose to allow access to certain choices that affect its operation. This information is accessed in a structured manner using interfaces derived from the **Policy** interface defined in the CORBA module. A CORBA service does not have to use this method of accessing operating options, but may choose to do so. The *Security Service* in particular uses this technique for associating *Security Policy* with objects in the system.

```
module CORBA {
    typedef unsigned long PolicyType;

    // Basic IDL definition
    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };

    typedef sequence <Policy> PolicyList;
    typedef sequence <PolicyType> PolicyTypeSeq;
};
```

PolicyType defines the type of **Policy** object. In general the constant values that are allocated are defined in conjunction with the definition of the corresponding **Policy** object. The values of **PolicyTypes** for policies that are standardized by OMG are allocated by OMG. Additionally, vendors may reserve blocks of 4096 PolicyType values identified by a 20 bit *Vendor PolicyType Valueset ID* (**VPVID**) for their own use.

PolicyType which is an unsigned long consists of the 20-bit **VPVID** in the high order 20 bits, and the vendor assigned policy value in the low order 12 bits. The **VPVID**s *0 through \xf* are reserved for OMG. All values for the standard **PolicyTypes** are allocated within this range by OMG. Additionally, the **VPVID**s *\xfffff* is reserved for experimental use and **OMGVMCID** is reserved for OMG use. These will not be allocated to anybody. Vendors can request allocation of **VPVID** by sending mail to tag-*request@omg.org*.

When a **VMCID** is allocated to a vendor automatically the same value of **VPVID** is reserved for the vendor and vice versa. So once a vendor gets either a **VMCID** or a **VPVID** registered they can use that value for both their minor codes and their policy types.

10.2.1.1 Copy

```
    Policy copy();
```

Return Value

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain, or object.

10.2.1.2 Destroy

```
    void destroy();
```

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

Exception(s)

CORBA::NO_PERMISSION - raised when the policy object determines that it cannot be destroyed.

10.2.1.3 Policy_type

```
    readonly attribute policy_type
```

Return Value

This readonly attribute returns the constant value of type **PolicyType** that corresponds to the type of the **Policy** object.

10.2.2 Creation of Policy Objects

A generic ORB operation for creating new instances of Policy objects is provided as described in this section.

```
module CORBA {

    typedef short PolicyErrorCode;
    const PolicyErrorCode BAD_POLICY = 0;
    const PolicyErrorCode UNSUPPORTED_POLICY = 1;
    const PolicyErrorCode BAD_POLICY_TYPE = 2;
    const PolicyErrorCode BAD_POLICY_VALUE = 3;
    const PolicyErrorCode UNSUPPORTED_POLICY_VALUE = 4;

    exception PolicyError {PolicyErrorCode reason;};

    interface ORB {

        .....

        Policy create_policy(
            in PolicyType type,
            in any val
        ) raises(PolicyError);
    };
};
```

10.2.2.1 PolicyErrorCode

A request to create a **Policy** may be invalid for the following reasons:

BAD_POLICY - the requested **Policy** is not understood by the ORB.

UNSUPPORTED_POLICY - the requested **Policy** is understood to be valid by the ORB, but is not currently supported.

BAD_POLICY_TYPE - The type of the value requested for the **Policy** is not valid for that **PolicyType**.

BAD_POLICY_VALUE - The value requested for the **Policy** is of a valid type but is not within the valid range for that type.

UNSUPPORTED_POLICY_VALUE - The value requested for the **Policy** is of a valid type and within the valid range for that type, but this valid value is not currently supported.

10.2.2.2 PolicyError

```
exception PolicyError {PolicyErrorCode reason;};
```

PolicyError exception is raised to indicate problems with parameter values passed to the **ORB::create_policy** operation. Possible reasons are described above.

10.2.2.3 Create_policy

The ORB operation **create_policy** can be invoked to create new instances of policy objects of a specific type with specified initial state. If **create_policy** fails to instantiate a new **Policy** object due to its inability to interpret the requested type and content of the policy, it raises the PolicyError exception with the appropriate reason as described in Section 10.2.2.1, "PolicyErrorCode," on page 159.

```
Policy create_policy(
    in PolicyType type,
    in any val
) raises(PolicyError);
```

Parameter(s)

type - the **PolicyType** of the policy object to be created.

val - the value that will be used to set the initial state of the **Policy** object that is created.

ReturnValue

Reference to a newly created **Policy** object of type specified by the **type** parameter and initialized to a state specified by the **val** parameter.

Exception(s)

PolicyError - raised when the requested policy is not supported or a requested initial state for the policy is not supported.

When new policy types are added to CORBA or CORBA Services specification, it is expected that the IDL type and the valid values that can be passed to **create_policy** also be specified.

10.3 Management of Policies

10.3.1 Client Side Policy Management

Client-side Policy management is performed through operations accessible in the following contexts:

- ORB-level Policies - A locality-constrained **PolicyManager** is accessible through the ORB interface. This **PolicyManager** has operations through which a set of Policies can be applied and the current overriding Policy settings can be obtained. Policies applied at the ORB level override any system defaults. The ORB's **PolicyManager** is obtained through an invocation of **ORB::resolve_initial_references**, specifying an identifier of "ORBPolicyManager."
- Thread-level Policies - A standard **PolicyCurrent** is defined with operations for the querying and applying of quality of service values specific to a thread. Policies applied at the thread level override any system defaults or values set at

the ORB level. The locality-constrained **PolicyCurrent** is obtained through an invocation of **ORB::resolve_initial_references**, specifying an identifier of "PolicyCurrent." When accessed from a newly spawned thread, the **PolicyCurrent** initially has no overridden policies. The **PolicyCurrent** also has no overridden values when a POA with **ThreadPolicy** of **ORB_CONTROL_MODEL** dispatches an invocation to a servant. Each time an invocation is dispatched through a **SINGLE_THREAD_MODEL** POA, the thread-level overrides are reset to have no overridden values.

- Object-level Policies - Operations are defined on the base Object interface through which a set of Policies can be applied. Policies applied at the Object level override any system defaults or values set at the ORB or Thread levels. In addition, accessors are defined for querying the current *overriding* Policies set at the Object level, and for obtaining the current *effective client-side* Policy of a given **PolicyType**. The e*ffective client-side* **Policy** is the value of a **PolicyType** that would be in effect if a request were made. This is determined by checking for overrides at the Object level, then at the Thread level, and finally at the ORB level. If no overriding policies are set at any level, the system-dependent default value is returned. Portable applications are expected to override the ORB-level policies since default values are not specified in most cases.

10.3.2 Server Side Policy Management

Server-side Policy management is handled by associating Policy objects with a POA. Since all policy objects are derived from interface **Policy**, those that are applicable to server-side behavior can be passed as arguments to **POA::create_POA**. Any such Policies that affect the behavior of requests (and therefore must be accessible to the ORB at the client side) are exported within the Object references that the POA creates. It is clearly noted in a POA **Policy** definition when that **Policy** is of interest to the Client. For those policies that can be exported within an Object reference, the absence of a value for that policy type implies that the target supports any legal value of that **PolicyType**.

Most Policies are appropriate only for management at either the Server or Client, but not both. For those Policies that can be established at the time of Object reference creation (through POA Policies) and overridden by the client (through overrides set at the ORB, thread, or Object reference scopes), reconciliation is done on a per-Policy basis. Such Policies are clearly noted in their definitions and describe the mechanism of reconciliation between the Policies that are set by the POA and overridden in the client. Furthermore, obtaining the effective **Policy** of some PolicyTypes requires evaluating the effective **Policy** of other types of Policies. Such hierarchical **Policy** definitions are also noted clearly when used.

At the Thread and ORB scopes, the common operations for querying the current set of policies and for overriding these settings are encapsulated in the **PolicyManager** interface.

10.3.3 Policy Management Interfaces

```
module CORBA {

    local interface PolicyManager {

        PolicyList get_policy_overrides(in PolicyTypeSeq ts);

        void set_policy_overrides(
                in PolicyList           policies,
                in SetOverrideType      set_add
        ) raises (InvalidPolicies);
    };

    local interface PolicyCurrent : PolicyManager, Current {
    };
};
```

10.3.3.1 interface PolicyManager

The **PolicyManager** operations are used for setting and accessing **Policy** overrides at a particular scope. For example, an instance of the **PolicyCurrent** is used for specifying **Policy** overrides that apply to invocations from that thread (unless they are overridden at the Object scope as described in Section 10.3.1, “Client Side Policy Management,” on page 160).

get_policy_overrides

```
    PolicyList get_policy_overrides(in PolicyTypeSeq ts);
```

Parameter

ts

a sequence of overridden policy types identifying the policies that are to be retrieved.

Return Value

policy list

The list of overridden policies of the types specified by ts.

Exceptions

none

Returns a **PolicyList** containing the overridden Polices for the requested PolicyTypes. If the specified sequence is empty, all **Policy** overrides at this scope will be returned. If none of the requested PolicyTypes are overridden at the target **PolicyManager**, an empty sequence is returned. This accessor returns only those **Policy** overrides that have been set at the specific scope corresponding to the target **PolicyManager** (no evaluation is done with respect to overrides at other scopes).

set_policy_overrides

```
    void set_policy_overrides(
            in PolicyList           policies,
            in SetOverrideType      set_add
    ) raises (InvalidPolicies);
```

Parameter

policies

A sequence of **Policy** objects that are to be associated with the **PolicyManager** object. If the sequence contains two or more **Policy** objects with the same **PolicyType** value, the operation raises the standard system exception BAD_PARAM with standard minor code 30.

set_add

Whether the association is in addition to (**ADD_OVERRIDE**) or as a replacement of (**SET_OVERRIDE**) any existing overrides already associated with the **PolicyManager** object.

If the value of this parameter is **SET_OVERRIDE**, the supplied **policies** completely replace all existing overrides associated with the **PolicyManager** object. If the value of this parameter is **ADD_OVERRIDE**, the supplied **policies** are added to the existing overrides associated with the **PolicyManager** object, except that if a supplied **Policy** object has the same **PolicyType** value as an existing override, the supplied **Policy** object replaces the existing override.

Return Value

none.

Exceptions

InvalidPolicies

A list of indices identifying the position in the input policies list that are occupied by invalid policies.

Modifies the current set of overrides with the requested list of **Policy** overrides. The first parameter policies is a sequence of references to **Policy** objects. The second parameter **set_add** of type **SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**ADD_OVERRIDE**) in the **PolicyManager**, or they should be added to a clean **PolicyManager** free of any other overrides (**SET_OVERRIDE**). Invoking **set_policy_overrides** with an empty sequence of policies and a mode of **SET_OVERRIDE** removes all overrides from a **PolicyManager**. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception. If the request would put the set of overriding policies for the target **PolicyManager** in an inconsistent state, no policies are changed or added, and the exception InvalidPolicies is raised. There is no evaluation of compatibility with policies set within other **PolicyManagers**.

10.3.3.2 interface PolicyCurrent

This specific **PolicyManager** provides access to policies overridden at the Thread scope. A reference to a thread's **PolicyCurrent** is obtained through an invocation of **CORBA::ORB::resolve_initial_references**.

10.4 Messaging Quality of Service

The Messaging module contains the IDL that the programmer uses to define Qualities of Service specific to CORBA messaging.

Note – Except where defaults are noted, this specification does not state required default values for the following Qualities of Service. Application code must explicitly set its ORB-level Quality of Service to ensure portability across ORB products.

```
module Messaging {

    typedef short RebindMode;
    const RebindMode TRANSPARENT =              0;
    const RebindMode NO_REBIND =                1;
    const RebindMode NO_RECONNECT =             2;
```

```
typedef short SyncScope;
const SyncScope SYNC_NONE =                 0;
const SyncScope SYNC_WITH_TRANSPORT =       1;
const SyncScope SYNC_WITH_SERVER =          2;
const SyncScope SYNC_WITH_TARGET =          3;

// Rebind Policy (default = TRANSPARENT)
const CORBA::PolicyType REBIND_POLICY_TYPE = 23;
local interface RebindPolicy : CORBA::Policy {
    readonly attribute RebindMode     rebind_mode;
};

// Synchronization Policy (default = SYNC_WITH_TRANSPORT)
const CORBA::PolicyType SYNC_SCOPE_POLICY_TYPE = 24;
    local interface SyncScopePolicy : CORBA::Policy {
    readonly attribute SyncScope      synchronization;
};

const CORBA::PolicyType REQUEST_PRIORITY_POLICY_TYPE = 25;

const CORBA::PolicyType REPLY_PRIORITY_POLICY_TYPE = 26;
// Timeout Policies
const CORBA::PolicyType
            REQUEST_START_TIME_POLICY_TYPE = 27;

const CORBA::PolicyType REQUEST_END_TIME_POLICY_TYPE = 28;
local interface RequestEndTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT end_time;
};

const CORBA::PolicyType REPLY_START_TIME_POLICY_TYPE = 29;

const CORBA::PolicyType REPLY_END_TIME_POLICY_TYPE = 30;
local interface ReplyEndTimePolicy : CORBA::Policy {
    readonly attribute TimeBase::UtcT end_time;
};

const CORBA::PolicyType
            RELATIVE_REQ_TIMEOUT_POLICY_TYPE = 31;
local interface RelativeRequestTimeoutPolicy : CORBA::Policy {
    readonly attribute TimeBase::TimeT relative_expiry;
};

const CORBA::PolicyType
            RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
local interface RelativeRoundtripTimeoutPolicy : CORBA::Policy {
    readonly attribute TimeBase::TimeT relative_expiry;
};

const CORBA::PolicyType ROUTING_POLICY_TYPE = 33;
const CORBA::PolicyType MAX_HOPS_POLICY_TYPE = 34;
```

```
    const CORBA::PolicyType QUEUE_ORDER_POLICY_TYPE = 35;
};
```

10.4.1 Rebind Support

Rebind support discussed in this section refers to the act of rebinding an object reference that has already been bound once. The policies discussed here do not affect the initial binding of an object reference.

10.4.1.1 typedef short RebindMode

Describes the level of transparent rebinding that may occur during the course of an invocation on an Object. Values of type **RebindMode** are used in conjunction with a **RebindPolicy**, as described in Section 10.4.1.2, "interface RebindPolicy," on page 165. All non-negative values are reserved for use in OMG specifications. Any negative value of **RebindMode** is considered a vendor extension.

- **TRANSPARENT** - allows the ORB to silently handle object-forwarding and necessary reconnection during the course of making a remote request. This is equivalent to the only defined *CORBA* ORB behavior.
- **NO_REBIND** - allows the ORB to silently handle reopening of closed connections while making a remote request, but prevents any transparent object-forwarding that would cause a change in client-visible effective QoS policies. When this policy is in effect, only explicit rebinding (through **CORBA::Object::validate_connection**) is allowed.
- **NO_RECONNECT** - prevents the ORB from silently handling object-forwards or the reopening of closed connections. When this policy is in effect, only explicit rebinding and reconnection (through **CORBA::Object::validate_connection**) is allowed.

10.4.1.2 interface RebindPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate whether the ORB may transparently rebind once successfully *bound* to a target. For GIOP-based protocols an object reference is considered bound once it is in a state where a **LocateRequest** message would result in a **LocateReply** message with status **OBJECT_HERE**. If the effective Policy of this type has a **rebind_mode** value of **TRANSPARENT** (always the default and the only valid value in *CORBA*), the ORB will silently handle any subsequent **LocateReply** messages with **OBJECT_FORWARD** status or Reply messages with **LOCATION_FORWARD** status. The effective policies of other types for this object reference may change from invocation to invocation. If the effective Policy of this type has a **rebind_mode** value of **NO_REBIND**, the ORB will raise a REBIND system exception if any rebind handling would cause a client-visible change in policies. This could happen under the following circumstances:

- The client receives a **LocateReply** message with an **OBJECT_FORWARD** status and a new IOR that has policy requirements incompatible with the effective policies currently in use.
- The client receives a Reply message with **LOCATION_FORWARD** status and a new IOR that has policy requirements incompatible with the effective policies currently in use.

If the effective Policy of this type has a **rebind_mode** value of **NO_RECONNECT**, the ORB will raise a REBIND system exception if any rebind handling would cause a client-visible change in policies, or if a new connection must be opened. This includes the reopening of previously closed connections as well as the opening of new connections if the target address changes (for example, due to a **LOCATION_FORWARD** reply). For connectionless protocols, the meaning of this effective policy must be specified, or it must be defined that **NO_RECONNECT** is an equivalent to **NO_REBIND**. Regardless of the effective **RebindPolicy**, rebind or reconnect can always be explicitly requested through an invocation of **CORBA::Object::validate_connection**. When instances of **RebindPolicy** are created, a value of type **RebindMode** is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RebindPolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REBIND_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **RebindMode**.

10.4.2 Synchronization Scope

10.4.2.1 typedef short SyncScope

Describes the level of synchronization for a request with respect to the target. Values of type **SyncScope** are used in conjunction with a **SyncScopePolicy**, as described in Section 10.4.2.2, “interface SyncScopePolicy,” on page 166, to control the behavior of oneway operations. All non-negative values are reserved for use in OMG specifications. Any negative value of **SyncScope** is considered a vendor extension.

- **SYNC_NONE** - equivalent to one allowable interpretation of *CORBA* oneway operations. The ORB returns control to the client (e.g., returns from the method invocation) before passing the request message to the transport protocol. The client is guaranteed not to block. Since no reply is returned from the server, no location-forwarding can be done with this level of synchronization.

- **SYNC_WITH_TRANSPORT** - equivalent to one allowable interpretation of *CORBA* oneway operations. The ORB returns control to the client only after the transport has accepted the request message. This in itself gives no guarantee that the request will be delivered, but in conjunction with knowledge of the characteristics of the transport may provide the client with a useful degree of assurance. For example, for a direct message over TCP, **SYNC_WITH_TRANSPORT** is not a stronger guarantee than **SYNC_NONE**. However, for a store-and-forward transport, this QoS provides a high level of reliability. Since no reply is returned from the server, no location-forwarding can be done with this level of synchronization.

- **SYNC_WITH_SERVER** - the server-side ORB sends a reply before invoking the target implementation. If a reply of **NO_EXCEPTION** is sent, any necessary location-forwarding has already occurred. Upon receipt of this reply, the client-side ORB returns control to the client application. This form of guarantee is useful where the reliability of the network is substantially lower than that of the server. The client blocks until all location-forwarding has been completed. For a server using a POA, the reply would be sent after invoking any ServantManager, but before delivering the request to the target Servant.

- **SYNC_WITH_TARGET** - equivalent to a synchronous, non-oneway operation in *CORBA*. The server-side ORB shall only send the reply message after the target has completed the invoked operation. Note that any **LOCATION_FORWARD** reply will already have been sent prior to invoking the target and that a **SYSTEM_EXCEPTION** reply may be sent at anytime (depending on the semantics of the exception). Even though it was declared oneway, the operation actually has the behavior of a synchronous operation. This form of synchronization guarantees that the client knows that the target has seen and acted upon a request. As with *CORBA*, only with this highest level of synchronization can the OTS be used. Any operations invoked with lesser synchronization precludes the target from participating in the client’s current transaction.

10.4.2.2 interface SyncScopePolicy

This interface is a local object derived from **CORBA::Policy**. It is applied to oneway operations to indicate the synchronization scope with respect to the target of that operation request. It is ignored when any non-oneway operation is invoked. This policy is also applied when the DII is used with a flag of **INV_NO_RESPONSE** since the implementation of the DII is not required to consult an interface definition to determine if an operation is declared oneway. The default value of this Policy is not defined. Applications must explicitly set an ORB-level **SyncScopePolicy** to ensure portability across ORB implementations. When instances of **SyncScopePolicy** are created, a value of type **Messaging::SyncScope** is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. The client's **SyncScopePolicy** is propagated within a request in the RequestHeader's **response_flags** as described in GIOP Request Header.

10.4.3 Request and Reply Timeout

This specification describes the lifetime of requests and replies in terms of the structured type from the CORBA Time Service Specification. This describes time as a 64-bit value, which is the number of 100 nano-seconds from 15 October 1582 00:00, along with inaccuracy and time zone information.

10.4.3.1 interface RequestEndTimePolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the time after which a request may no longer be delivered to its target. This policy is applied to both synchronous and asynchronous invocations. When instances of **RequestEndTimePolicy** are created, a value of type **TimeBase::UtcT** is containing an absolute time passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RequestEndTimePolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REQUEST_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **TimeBase::UtcT**.

The client ORB, all routers and the target ORB shall check to see if the end time specified in the **RequestEndTimePolicy** associated with a request has expired and the request is yet to be delivered to the target. If so, it shall the discard the request and return the system exception **TIMEOUT** with standard minor code 2.

10.4.3.2 interface ReplyEndTimePolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the time after which a reply may no longer be obtained or returned to the client. This policy is applied to both synchronous and asynchronous invocations. When instances of **ReplyEndTimePolicy** are created, a value of type **TimeBase::UtcT** containing an absolute time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **ReplyEndTimePolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REPLY_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing a **TimeBase::UtcT**.

The client ORB, all routers and the target ORB shall check to see if the end time specified in the **ReplyEndTimePolicy** associated with a request has expired and a reply has not yet been delivered to the client. If so, it shall the discard the reply and return the system exception **TIMEOUT** with standard minor code 3.

10.4.3.3 interface RelativeRequestTimeoutPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the relative amount of time for which a Request may be delivered. After this amount of time the Request is cancelled. This policy is applied to both synchronous and asynchronous invocations. If asynchronous invocation is used, this policy only limits the amount of time during which the request may be processed. Assuming the request completes within the specified timeout, the reply will never be discarded due to timeout. When instances of **RelativeRequestTimeoutPolicy** are created, a value of type **TimeBase::TimeT** containing a relative time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RelativeRequestTimeoutPolicy** is propagated within a

PolicyValue in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REQUEST_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing the **relative_expiry** converted into a **TimeBase::UtcT** end time (as in the case of **RequestEndTimePolicy**).

Since a **RelativeRequestTimeoutPolicy** is converted to a **RequestEndTimePolicy** before transmitting the request to the target ORB, see section 10.4.3.1 for the required behavior of an ORB or router when the timeout expires.

10.4.3.4 interface RelativeRoundtripTimeoutPolicy

This interface is a local object derived from **CORBA::Policy**. It is used to indicate the relative amount of time for which a Request or its corresponding Reply may be delivered. After this amount of time, the Request is cancelled (if a response has not yet been received from the target) or the Reply is discarded (if the Request had already been delivered and a Reply returned from the target). This policy is applied to both synchronous and asynchronous invocations.

When instances of **RelativeRoundtripTimeoutPolicy** are created, a value of type **TimeBase::TimeT** containing a relative time is passed to **CORBA::ORB::create_policy**. This policy is only applicable as a client-side override. When an instance of **RelativeRoundtripTimeoutPolicy** is propagated within a **PolicyValue** in an **INVOCATION_POLICIES** Service Context, the **ptype** has value **REPLY_END_TIME_POLICY_TYPE** and the **pvalue** is a CDR encapsulation containing the **relative_expiry** converted into a **TimeBase::UtcT** end time (as in the case of **ReplyEndTimePolicy**).

Since a **RelativeRoundtripTimeoutPolicy** is converted to a **ReplyEndTimePolicy** before transmitting the request to the target ORB, see section 10.4.3.2 for the required behavior of an ORB or router when the timeout expires.

10.5 Consolidated IDL

10.5.1 Messaging Module

The following module has been added by CORBA Messaging:

```
// IDL
// File: Messaging.idl
#ifndef _MESSAGING_IDL_
#define _MESSAGING_IDL_

import ::CORBA;
import ::IOP;
import ::TimeBase;
module Messaging {
    typeprefix Messaging “omg.org”;
    //
    // Messaging Quality of Service
    //

    typedef short RebindMode;
    const RebindMode TRANSPARENT =              0;
    const RebindMode NO_REBIND =                1;
    const RebindMode NO_RECONNECT =             2;
    typedef short SyncScope;
    const SyncScope SYNC_NONE =                    0;
    const SyncScope SYNC_WITH_TRANSPORT =   1;
    const SyncScope SYNC_WITH_SERVER =         2;
    const SyncScope SYNC_WITH_TARGET =         3;

    //
    // Locally-Constrained Policy Objects
    //

    // Rebind Policy (default = TRANSPARENT)
    const CORBA::PolicyType REBIND_POLICY_TYPE = 23;
    interface RebindPolicy : CORBA::Policy {
        readonly attribute RebindMode     rebind_mode;
    };

    // Synchronization Policy (default = SYNC_WITH_TRANSPORT)
    const CORBA::PolicyType SYNC_SCOPE_POLICY_TYPE = 24;
    interface SyncScopePolicy : CORBA::Policy {
        readonly attribute SyncScope       synchronization;
    };
    const CORBA::PolicyType REQUEST_PRIORITY_POLICY_TYPE = 25;
    const CORBA::PolicyType REPLY_PRIORITY_POLICY_TYPE = 26;

    // Timeout Policies
    const CORBA::PolicyType
                        REQUEST_START_TIME_POLICY_TYPE = 27;
    const CORBA::PolicyType REQUEST_END_TIME_POLICY_TYPE = 28;
    interface RequestEndTimePolicy : CORBA::Policy {
        readonly attribute TimeBase::UtcT end_time;
    };

    const CORBA::PolicyType REPLY_START_TIME_POLICY_TYPE = 29;
    const CORBA::PolicyType REPLY_END_TIME_POLICY_TYPE = 30;
    interface ReplyEndTimePolicy : CORBA::Policy {
        readonly attribute TimeBase::UtcT end_time;
    };

    const CORBA::PolicyType
                        RELATIVE_REQ_TIMEOUT_POLICY_TYPE = 31;
    interface RelativeRequestTimeoutPolicy : CORBA::Policy {
        readonly attribute TimeBase::TimeT relative_expiry;
    };

    const CORBA::PolicyType
                        RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
    interface RelativeRoundtripTimeoutPolicy : CORBA::Policy {
        readonly attribute TimeBase::TimeT relative_expiry;
    };
};

module CORBA {
```

```
local interface PolicyManager {

      PolicyList get_policy_overrides(in PolicyTypeSeq ts);

      void set_policy_overrides(
                  in PolicyList             policies,
                  in SetOverrideType        set_add
      ) raises (InvalidPolicies);
};

local interface PolicyCurrent : PolicyManager, Current {};
```

11 The Portable Object Adapter

Compliance

Conformant implementations of the CORBA/e Compact Profile must comply with clauses 8.1, 8.2, 8.3 and 8.4.1 of this chapter.

Conformant implementations of the CORBA/e Micro Profile must support a single root POA and must comply with clauses 8.1, 8.2, 8.3 (except 8.3.4.1, 8.3.4.2, 8.3.4.4, 8.3.4.9, and 8.3.4.12) and 8.4.2 of this chapter.

11.1 Overview

The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementations that can provide consistent service for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes.
- Allow a single servant to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.
- Provide support for transient objects with minimal programming effort and overhead.
- Allow object implementations to be maximally responsible for an object's behavior. Specifically, an implementation can control an object's behavior by establishing the datum that defines an object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing the code that will be executed in response to requests, and determining whether or not the object exists at any point in time.
- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information with objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compilers/.

11.2 Abstract Model Description

The POA interfaces described in this chapter imply a particular abstract computational model. This section presents that model and defines terminology and basic concepts that will be used in subsequent sections.

Note – The CORBA/*e* Profiles restrict themselves to the more static features; the following description is a subset of the abstract model for the POA supported by other CORBA specifications.

This section provides the rationale for the POA design, describes some of its intended uses, and provides a background for understanding the interface descriptions.

11.2.1 Model Components

The model supported by the POA is a specialization of the general object model described in the OMA guide. Most of the elements of the CORBA object model are present in the model described here, but there are some new components, and some of the names of existing components are defined more precisely than they are in the CORBA object model. The abstract model supported by the POA has the following components:

- *Client*—A client is a computational context that makes requests on an object through one of its references.
- *Server*—A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process. Note that *client* and *server* are roles that programs play with respect to a given object. A program that is a client for one object may be the server for another. The same process may be both client and server for a single object.
- *Object*—In this discussion, we use *object* to indicate a CORBA object in the abstract sense, that is, a programming entity with an identity, an interface, and an implementation. From a client's perspective, the object's identity is encapsulated in the object's reference. This specification defines the server's view of object identity, which is explicitly managed by object implementations through the POA interface.
- *Servant*—A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests on its references will be targeted at) multiple servants.
- *Object Id*—An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.
- Note that the Object Id defined in this specification is a mechanical device used by an object implementation to correlate incoming requests with references it has previously created and exposed to clients. It does not constitute a unique logical identity for an object in any larger sense. The assignment and interpretation of Object Id values is primarily the responsibility of the application developer, although the **SYSTEM_ID** policy enables the POA to generate Object Id values for the application.
- *Object Reference*—An object reference in this model is the same as in the CORBA object model. This model implies, however, that a reference specifically encapsulates an Object Id and a POA identity.

  Note that a concrete reference in a specific ORB implementation will contain more information, such as the location of the server and POA in question. For example, it might contain the full name of the POA (the names of all POAs starting from the root and ending with the specific POA). The reference might not, in fact, actually contain the Object Id, but instead contain more compact values managed by the ORB that can be mapped to the Object Id. This is a description of the abstract information model implied by the POA. Whatever encoding is used to represent the POA name and the Object Id must not restrict the ability to use any legal character in a POA name or any legal octet in an Object Id.
- *POA*—A POA is an identifiable entity within the context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.

- *Policy*—A Policy is an object associated with a POA by an application in order to specify a characteristic shared by the objects implemented in that POA. This specification defines policies controlling the POA's threading model as well as a variety of other options related to the management of objects. Other specifications may define other policies that affect how an ORB processes requests on objects implemented in the POA.
- *POA Manager*—A POA manager is an object that encapsulates the processing state of one or more POAs. Using operations on a POA manager, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the POA manager to deactivate the POAs.

11.2.2 Model Architecture

This section describes the architecture of the abstract model implied by the POA, and the interactions between various components. The ORB is an abstraction visible to both the client and server. The POA is an object visible to the server. User-supplied implementations are registered with the POA (this statement is a simplification; more detail is provided below). Clients hold references upon which they can make requests. The ORB, POA, and implementation all cooperate to determine which servant the operation should be invoked on, and to perform the invocation.

Figure 11.1 shows the detail of the relationship between the POA and the implementation. Ultimately, a POA deals with an Object Id and an active servant. By *active servant*, we mean a programming object that exists in memory and has been presented to the POA with one or more associated object identities.

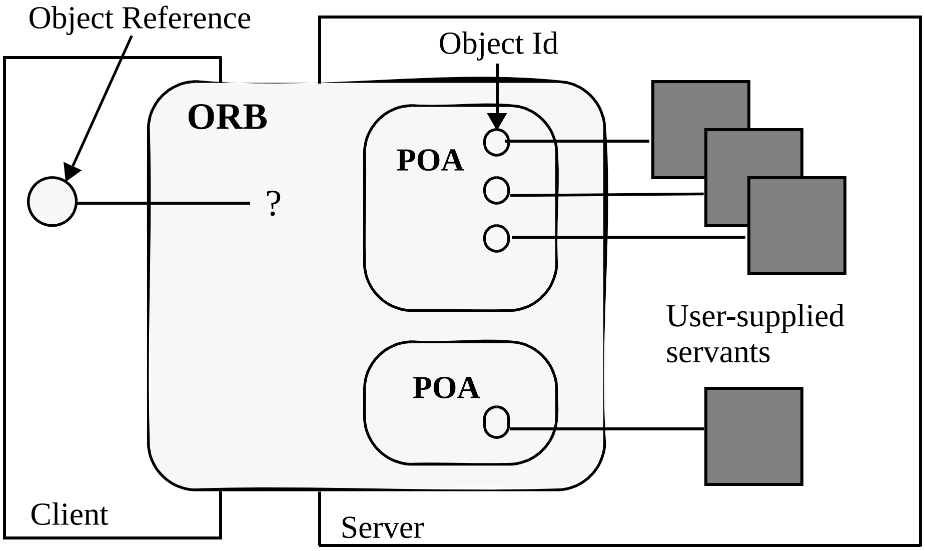

Figure 11.1- Abstract POA Model

The POA maintains a map, labeled *Active Object Map*, that associates Object Ids with active servants, each association constituting an active object.

In this specification, the term *active* is applied equally to servants, Object Ids, and objects. An object is active in a POA if the POA's Active Object Map contains an entry that associates an Object Id with an existing servant. When this specification refers to *active Object Ids* and *active servants*, it means that the Object Id value or servant in question is part of an entry in the Active Object Map. An Object Id can appear in a POA's Active Object Map only once.

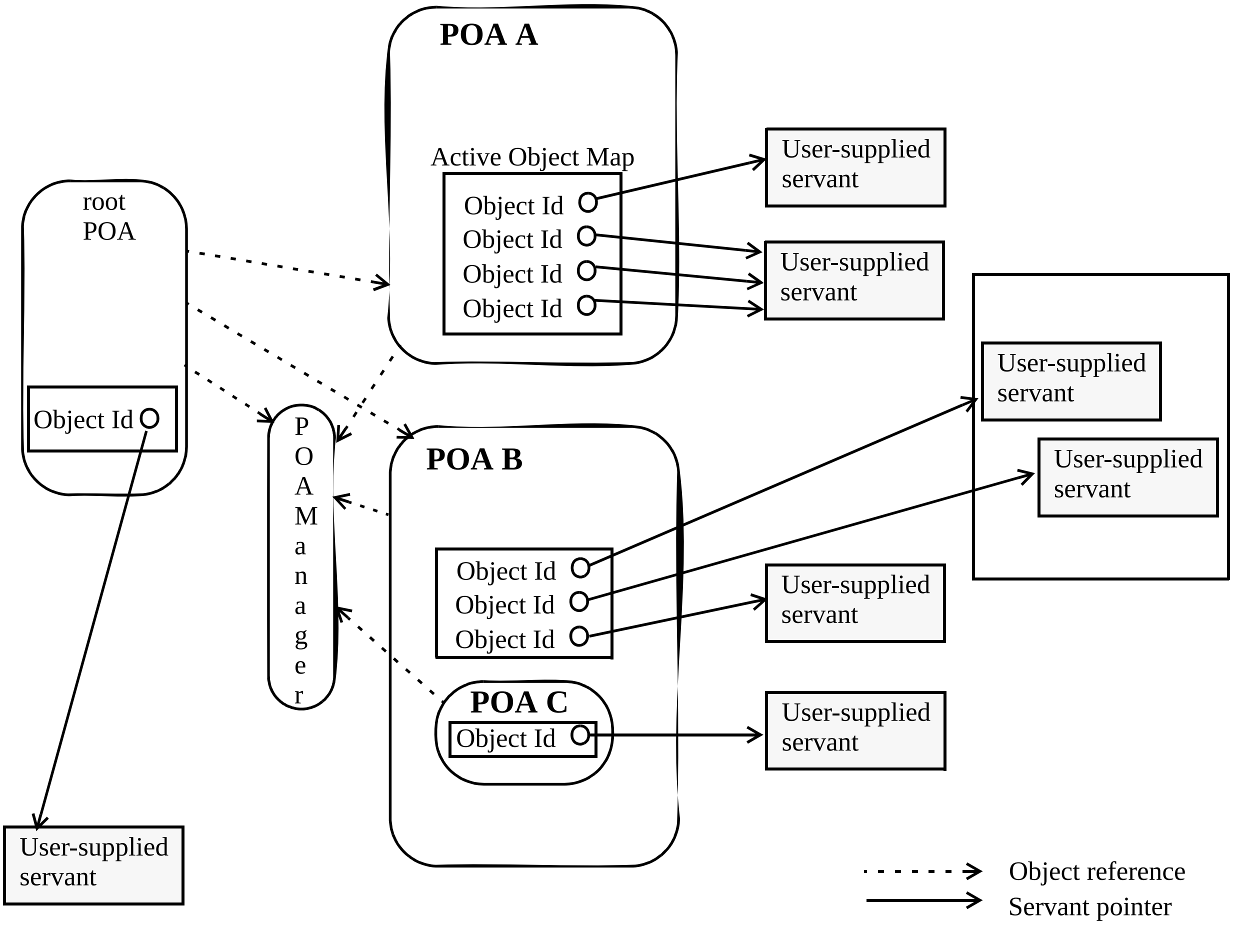

Figure 11.2- POA Architecture

11.2.3 POA Creation

To implement an object using the POA requires that the server application obtain a POA object. A distinguished POA object, called the *root POA*, is managed by the ORB and provided to the application using the ORB initialization interface under the initial object name “RootPOA.” The application developer can create objects using the root POA if those default policies are suitable. The root POA has the following policies.

- Thread Policy: **ORB_CTRL_MODEL**
- Lifespan Policy: **TRANSIENT**
- Object Id Uniqueness Policy: **UNIQUE_ID**
- Id Assignment Policy: **SYSTEM_ID**
- Servant Retention Policy: **RETAIN**

- Request Processing Policy: **USE_ACTIVE_OBJECT_MAP_ONLY**
- Implicit Activation Policy: **IMPLICIT_ACTIVATION**

The developer can also create new POAs. Creating a new POA allows the application developer to declare specific policy choices for the new POA. Creating new POAs also allows the application developer to partition the name space of objects, as Object Ids are interpreted relative to a POA.

A POA is created as a child of an existing POA using the **create_POA** operation on the parent POA. When a POA is created, the POA is given a name that must be unique with respect to all other POAs with the same parent.

POA objects are not persistent. No POA state can be assumed to be saved by the ORB. It is the responsibility of the server application to create and initialize the appropriate POA objects during server initialization.

Creating the appropriate POA objects is particularly important for persistent objects, objects whose existence can span multiple server lifetimes. To support an object reference created in a previous server process, the application must recreate the POA that created the object reference as well as all of its ancestor POAs. To ensure portability, each POA must be created with the same name as the corresponding POA in the original server process and with the same policies. (It is the user's responsibility to create the POA with these conditions.)

A portable server application can presume that there is no conflict between its POA names and the POA names chosen by other applications. It is the responsibility of the ORB implementation to provide a way to support this behavior.

Each distinct ORB created as the result of an **ORB_init** call in an application has its own separate root POA and POA namespace.

11.2.4 Reference Creation

Object references are created in servers. Once they are created, they may be exported to clients.

From this model's perspective, object references encapsulate object identity information and information required by the ORB to identify and locate the server and POA with which the object is associated (that is, in whose scope the reference was created.) References are created in the following ways:

- The server application may directly create a reference with the **create_reference** and **create_reference_with_id** operations on a POA object. These operations collect the necessary information to constitute the reference, either from information associated with the POA or as parameters to the operation. These operations only create a reference. In doing so, they bring the abstract object into existence, but do not associate it with an active servant.
- The server application may explicitly activate a servant, associating it with an object identity using the **activate_object** or **activate_object_with_id** operations. Once a servant is activated, the server application can map the servant to its corresponding reference using the **servant_to_reference** or **id_to_reference** operations.

Once a reference is created in the server, it can be made available to clients in a variety of ways. It can be advertised through the OMG Naming and Trading Services. It can be converted to a string via **ORB::object_to_string** and published in some way that allows the client to discover the string and convert it to a reference using **ORB::string_to_object**. It can be returned as the result of an operation invocation.

Once a reference becomes available to a client, that reference constitutes the identity of the object from the client's perspective. As long as the client program holds and uses that reference, requests made on the reference should be sent to the "same" object.

The states of servers and implementation objects are opaque to clients. This specification deals primarily with the view of the ORB from the server's perspective.

11.2.5 Object Activation States

At any point in time, a CORBA object may or may not be associated with an active servant.

The servant and its associated Object Id are entered into the Active Object Map of the appropriate POA. This type of activation can be accomplished in one of the following ways.

- The server application itself explicitly activates individual objects (via the **activate_object** or **activate_object_with_id** operations).

11.2.6 Request Processing

A request must be capable of conveying the Object Id of the target object as well as the identification of the POA that created the target object reference. When a client issues a request, the ORB first locates an appropriate server (perhaps starting one if needed) and then it locates the appropriate POA within that server.

Once the ORB has located the appropriate POA, it delivers the request to that POA. The POA looks in the Active Object Map to find out if there is a servant associated with the Object Id value from the request. If such a servant exists, the POA invokes the appropriate method on the servant.

If the POA didn't find a servant in the Active Object Map, the POA raises the OBJECT_NOT_EXIST system exception with standard minor code 2.

11.2.7 Multi-threading

The POA does not require the use of threads and does not specify what support is needed from a threads package.

The POA in CORBA/*e* supports a single model of threading when used in conjunction with multi-threaded ORB implementations; ORB controlled. The ORB controlled model of threading implies the ORB/POA controls the use of threads in the manner provided by the ORB. This model can also be used in environments that do not support threads.

In this model, the ORB is responsible for the creation, management, and destruction of threads used with one or more POAs.

There are no guarantees that the ORB and POA will do anything specific about dispatching requests across threads with a single POA. Therefore, a server programmer who wants to use one or more POAs within multiple threads must take on all of the serialization of access to objects within those threads.

There may be requests active for the same object being dispatched within multiple threads at the same time. The programmer must be aware of this possibility and code with it in mind.

11.2.8 Location Transparency

The POA supports location transparency for objects implemented using the POA. Unless explicitly stated to the contrary, all POA behavior described in this specification applies regardless of whether the client is local (same process) or remote. For example, like a request from a remote client, a request from a local client may cause object activation if the object is not active, block indefinitely if the target object’s POA is in the holding state, be rejected if the target object’s POA is in the discarding or inactive states, or be delivered to a thread-unaware object implementation. The Object Id and POA of the target object will also be available to the server via the **Current** object, regardless of whether the client is local or remote.

Note – The implication of these requirements on the ORB implementation is to require the ORB to mediate all requests to POA-based objects, even if the client is co-resident in the same process. This specification is not intended to change CORBAServices specifications that allow for behaviors that are not location transparent. This specification does not prohibit (nonstandard) POA extensions to support object behavior that is not location-transparent.

11.2.9 UML Description of PortableServer

The following diagrams were generated by an automated tool and then annotated with the cardinalities of the associations. They are intended to be an aid in comprehension to those who enjoy such representations. They are not normative.

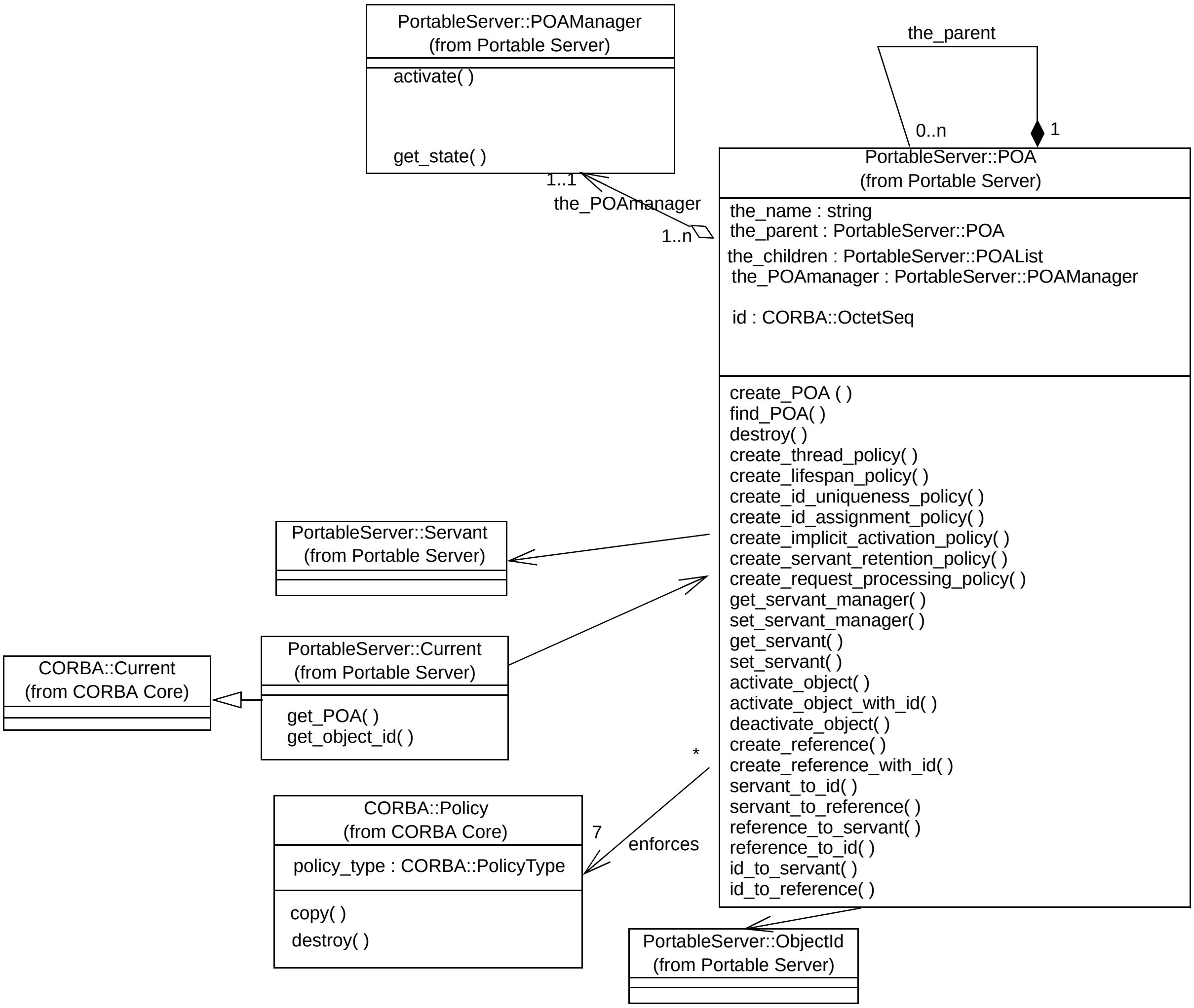

Figure 11.3- UML for main part of PortableServer

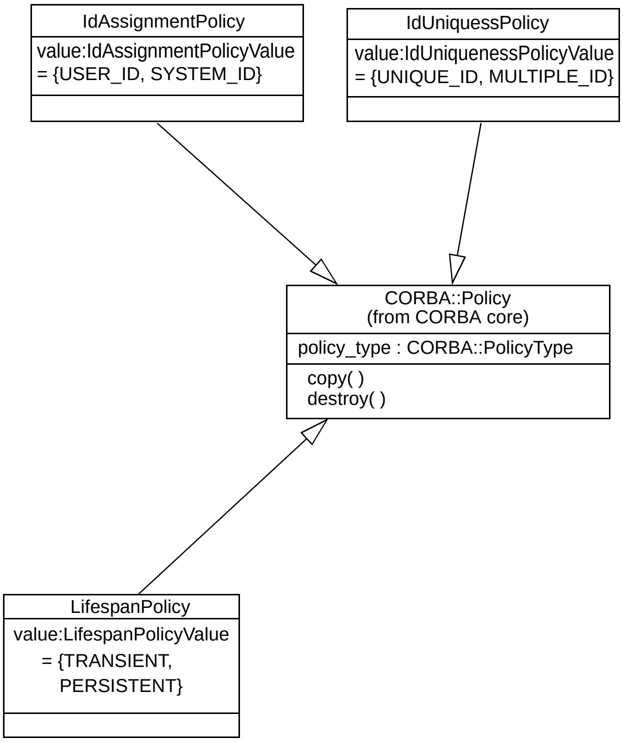

Figure 11.4- UML for PortableServer Policies

11.3 Interfaces

The POA-related interfaces are defined in a module separate from the **CORBA** module, the **PortableServer** module. It consists of these interfaces:

- POA
- POAManager
- ThreadPolicy
- LifespanPolicy
- IdUniquenessPolicy
- IdAssignmentPolicy
- Current

In addition, the POA defines the **Servant** native type.

All local objects specified in this chapter override the default behavior of the **Object::get_orb** operation and return the **ORB** that is associated with the root POA local object.

11.3.1 The Servant IDL Type

This specification defines a native type **PortableServer::Servant**. Values of the type **Servant** are programming-language-specific implementations of CORBA interfaces. Each language mapping must specify how **Servant** is mapped to the programming language data type that corresponds to an object implementation. The **Servant** type has the following characteristics and constraints.

- Values of type **Servant** are opaque from the perspective of CORBA application programmers. There are no operations that can be performed directly on them by user programs. They can be passed as parameters to certain POA operations. Some language mappings may allow **Servant** values to be implicitly converted to object references under appropriate conditions.
- Values of type **Servant** support a language-specific programming interface that can be used by the ORB to obtain a default POA for that servant. This interface is used only to support implicit activation. A language mapping may provide a default implementation of this interface that returns the root POA of a default ORB.
- Values of type Servant provide default implementations of the standard object reference operations **get_interface**, **is_a**, **repository_id**, and **non_existent**. These operations can be overridden by the programmer to provide additional behavior needed by the object implementation. The default implementations of **get_interface**, **repository_id,** and **is_a** operations use the most derived interface of a static servant or the most derived interface retrieved from a dynamic servant to perform the operation. The default implementation of the **non_existent** operation returns **FALSE**. These operations are invoked by the POA just like any other operation invocation, so the **PortableServer::Current** interface and any language-mapping-provided method of accessing the invocation context are available.
  - Values of type **Servant** must be testable for identity.
  - Values of type **Servant** have no meaning outside of the process context or address space in which they are generated.

11.3.2 POAManager Interface

Each POA object has an associated **POAManager** object. A POA manager may be associated with one or more POA objects. A POA manager encapsulates the processing state of the POAs it is associated with.

POAManager is a local interface.

11.3.2.1 Processing States

Note – The set of states and transitions supported by CORBA/*e* is deliberately restricted from those supported by other CORBA specifications.

A POA manager has three possible processing states :*inactive*, *active*, and *holding*. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs. Figure 11.5 on page 180 illustrates the processing states and the transitions between them. For simplicity of presentation, this specification sometimes describes these states as POA states, referring to the POA or POAs that have been associated with a particular POA manager. A POA manager is created in the *holding* state. The root POA is therefore initially in the *holding* state.

For simplicity in the figure and the explanation, operations that would not cause a state change are not shown. For example, if a POA is in “active” state, it does not change state due to an activate operation. Such operations complete successfully with no special notice.

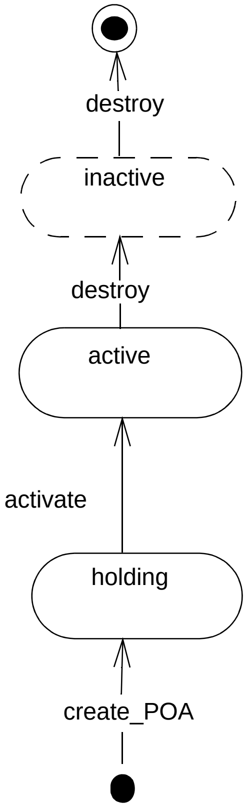

Figure 11.5- Processing States

Active State

When a POA manager is in the *active* state, the associated POAs will receive and start processing requests (assuming that appropriate thread resources are available). Note that even in the active state, a POA may need to queue requests depending upon the ORB implementation and resource limits. The number of requests that can be received and/or queued is an implementation limit. If this limit is reached, the POA should return a TRANSIENT system exception, with standard minor code 1, to indicate that the client should re-issue the request.

The POA enters the *active* state through the use of the **activate** operation when in the *holding* state.

Holding State

When a POA manager is in the *holding* state, the associated POAs will queue incoming requests. The number of requests that can be queued is an implementation limit. If this limit is reached, the POAs may discard requests and return the TRANSIENT system exception, with standard minor code 1, to the client to indicate that the client should reissue the request. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

A POA manager can legally transition from the *holding* state to the *active* state by calling the **activate** operation. A POA manager is created in the holding state.

Inactive State

The *inactive* state is entered when the associated POAs are to be shut down. When a POA manager is in the *inactive* state, the associated POAs will reject new requests. The rejection mechanism used is specific to the vendor. The GIOP location forwarding mechanism and CloseConnection message are examples of mechanisms that could be used to indicate the rejection. If the client is co-resident in the same process, the ORB could raise the OBJ_ADAPTER system exception, with standard minor code 1, to indicate that the object implementation is unavailable.

11.3.2.2 activate

```
void activate()
    raises (AdapterInactive);
```

This operation changes the state of the POA manager to *active*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *active* state enables the associated POAs to process requests.

11.3.2.3 get_state

```
enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

This operation returns the state of the POA manager.

11.3.3 POA Policy Objects

Interfaces derived from **CORBA::Policy** are used with the **POA::create_POA** operation to specify policies that apply to a POA. Policy objects are created using factory operations on any pre-existing POA, such as the root POA, or by a call to **ORB::create_policy**. Policy objects are specified when a POA is created. Policies may not be changed on an existing POA. Policies are not inherited from the parent POA. All **Policy** interfaces defined in this section are local interfaces.

Note – The behavior options represented by the POA policy values are subsetted from those available in other CORBA specifications. This is reflected in this specification by omitting some policy objects, when policy values are fixed to one value, or by restricting policy values, when appropriate.

11.3.3.1 CORBA/e Compact Profile

Thread Policy

CORBA/*e* supports the **ORB_CTRL_MODEL** behavior. The ORB is responsible for assigning requests for an ORB-controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.

Lifespan Policy

Objects with the **LifespanPolicy** interface are obtained using the **POA::create_lifespan_policy** operation and passed to the **POA::create_POA** operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied.

- **TRANSIENT** - The objects implemented in the **POA** cannot outlive the **POA** instance in which they are first created. Once the POA's **POAManager** enters the deactivated state, any requests received by this **POA** will cause the **POA** to raise an OBJECT_NOT_EXIST system exception with standard minor code 4.

- **PERSISTENT** - The objects implemented in the **POA** can outlive the process in which they are first created.
  - Persistent objects have a **POA** associated with them (the **POA** that created them). When the ORB receives a request on a persistent object, it first searches for the matching **POA**, based on the names of the **POA** and all of its ancestors.
  - Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this **POA**, and optionally to arrange for on-demand activation of a process implementing this **POA**.
  - **POA** names must be unique within their enclosing scope (the parent **POA**). A portable program can assume that **POA** names used in other processes will not conflict with its own **POA** names. A conforming CORBA implementation will provide a method for ensuring this property.

If no **LifespanPolicy** object is passed to **create_POA**, the lifespan policy defaults to **TRANSIENT**.

Object Id Uniqueness *Policy*

Objects with the **IdUniquenessPolicy** interface are obtained using the **POA::create_id_uniqueness_policy** operation and passed to the **POA::create_POA** operation to specify whether the servants activated in the created **POA** must have unique object identities. The following values can be supplied.

- **UNIQUE_ID** - Servants activated with that POA support exactly one Object Id.
- **MULTIPLE_ID** - a servant activated with that POA may support one or more Object Ids.

If no **IdUniquenessPolicy** is specified at POA creation, the default is **UNIQUE_ID**.

Id Assignment *Policy*

Objects with the **IdAssignmentPolicy** interface are obtained using the **POA::create_id_assignment_policy** operation and passed to the **POA::create_POA** operation to specify whether Object Ids in the created **POA** are generated by the application or by the ORB. The following values can be supplied.

- **USER_ID** - Objects created with that **POA** are assigned Object Ids only by the application.
- **SYSTEM_ID** - Objects created with that **POA** are assigned Object Ids only by the **POA**. If the **POA** also has the **PERSISTENT** policy, assigned Object Ids must be unique across all instantiations of the same POA.

If no **IdAssignmentPolicy** is specified at POA creation, the default is **SYSTEM_ID**.

Servant Retention *Policy*

CORBA/*e* supports the **RETAIN** behavior. The POA will retain active servants in its Active Object Map.

Request Processing Policy

CORBA/*e* supports the **USE_ACTIVE_OBJECT_MAP_ONLY** behavior. If the Object Id is not found in the Active Object Map, an OBJECT_NOT_EXIST system exception with standard minor code 2 is returned to the client.

Implicit Activation *Policy*

CORBA/*e* supports the **NO_IMPLICIT_ACTIVATION** behavior. The POA will not support implicit activation of servants.

11.3.3.2 CORBA/e Micro Profile

The CORBA/*e* Micro Profile supports the Root POA only. The policies for the root POA are:

Thread Policy

The CORBA/*e* Micro Profile root POA supports the **ORB_CTRL_MODEL** behavior. The ORB is responsible for assigning requests for an ORB- controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.

Lifespan Policy

The CORBA/*e* Micro Profile root POA supports the **TRANSIENT** behavior. The objects implemented in the **POA** cannot outlive the **POA** instance in which they are first created. Once the POA's **POAManager** enters the deactivated state, any requests received by this **POA** will cause the **POA** to raise an OBJECT_NOT_EXIST system exception with standard minor code 4.

Object Id Uniqueness *Policy*

The CORBA/*e* Micro Profile root POA supports the **UNIQUE_ID** behavior. Servants activated with that POA support exactly one Object Id.

Id Assignment *Policy*

The CORBA/*e* Micro Profile root POA supports the **SYSTEM_ID** behavior. Objects created with that **POA** are assigned Object Ids only by the **POA**.

Servant Retention *Policy*

The CORBA/*e* Micro Profile root POA supports the **RETAIN** behavior. The POA will retain active servants in its Active Object Map.

Request Processing Policy

The CORBA/*e* Micro Profile root POA supports the **USE_ACTIVE_OBJECT_MAP_ONLY** behavior. If the Object Id is not found in the Active Object Map, an OBJECT_NOT_EXIST system exception with standard minor code 2 is returned to the client.

Implicit Activation *Policy*

The CORBA/*e* Micro Profile root POA supports the **NO_IMPLICIT_ACTIVATION** behavior. The POA will not support implicit activation of servants.

11.3.4 POA Interface

A POA object manages the implementation of a collection of objects. The POA supports a name space for the objects, which are identified by Object Ids.

A POA also provides a name space for POAs. A POA is created as a child of an existing POA, which forms a hierarchy starting with the root POA.

The **POA** interface is a local interface.

11.3.4.1 create_POA

```
POA create_POA(
    in string               adapter_name,
    in POAManager           a_POAManager,
    in CORBA::PolicyList    policies)
    raises (AdapterAlreadyExists, InvalidPolicy
);
```

This operation creates a new POA as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA. If the target POA already has a child POA with the specified name, the AdapterAlreadyExists exception is raised.

If the **a_POAManager** parameter is null, a new **POAManager** object is created and associated with the new POA. Otherwise, the specified **POAManager** object is associated with the new POA. The **POAManager** object can be obtained using the attribute name **the_POAManager**.

The specified policy objects are associated with the POA and used to control its behavior. The policy objects are effectively copied before this operation returns, so the application is free to destroy them while the POA is in use. Policies are *not* inherited from the parent POA.

If any of the policy objects specified are not valid for the ORB implementation, if conflicting policy objects are specified, or if any of the specified policy objects require prior administrative action that has not been performed, an InvalidPolicy exception is raised containing the index in the policies parameter value of the first offending policy object.

Note – Creating a POA using a POA manager that is in the active state can lead to race conditions if the POA supports pre-existing objects, because the new POA may receive a request before its adapter activator, servant manager, or default servant have been initialized. These problems do not occur if the POA is created by an adapter activator registered with a parent of the new POA, because requests are queued until the adapter activator returns.

11.3.4.2 find_POA

```
POA find_POA(
    in string       adapter_name,
    in boolean      activate_it)
    raises (AdapterNonExistent
);
```

If the target **POA** is the parent of a child **POA** with the specified name (relative to the target **POA**), that child **POA** is returned. If a child **POA** with the specified name does not exist, the AdapterNonExistent exception is raised.

11.3.4.3 destroy

```
void destroy(
    in boolean      etherealize_objects,
    in boolean      wait_for_completion
);
```

This operation destroys the **POA** and all descendant **POAs**. All descendant **POA**s are destroyed (recursively) before the destruction of the containing **POA**. The **POA** so destroyed (that is, the **POA** with its name) may be re-created later in the same process.

When **destroy** is called the **POA** behaves as follows:

- The **POA** assumes the *discarding* state. Any further changes to the **POAManager**'s state do not affect this **POA**.
- The **POA** disables the **create_POA** operation. Subsequent calls to **create_POA** will result in a BAD_INV_ORDER system exception with standard minor code 17.
- The **POA** calls destroy on all of its immediate descendants.
- After all descendant **POA**s have been destroyed, the **POA** continues to process requests until there are no requests executing in the **POA**. At this point, apparent destruction of the **POA** has occurred.
- After destruction has become apparent, the **POA** may be re-created via either an AdapterActivator or a call to **create_POA**.

The **wait_for_completion** parameter is handled as follows:

- If **wait_for_completion** is **TRUE** and the current thread is not in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the destroy operation returns only after all active requests have completed and all invocations of **etherealize** have completed.
- If **wait_for_completion** is **TRUE** and the current thread is in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the BAD_INV_ORDER system exception with standard minor code 3 is raised and **POA** destruction does not occur.
- If **wait_for_completion** is **FALSE**, the **destroy** operation destroys the POA and its children but does not wait for active requests to complete. The **wait_for_completion** parameter is handled as defined above for each individual call (some callers may choose to block, while others may not).

11.3.4.4 Policy Creation Operations

```
LifespanPolicy create_lifespan_policy(
        in LifespanPolicyValue value);
IdUniquenessPolicy create_id_uniqueness_policy(
        in IdUniquenessPolicyValue value);
IdAssignmentPolicy create_id_assignment_policy(
        in IdAssignmentPolicyValue value);
```

These operations each return a reference to a policy object with the specified value. The application is responsible for calling the inherited **destroy** operation on the returned reference when it is no longer needed.

11.3.4.5 the_name

readonly attribute string the_name;

This attribute identifies the POA relative to its parent. This name is assigned when the POA is created. The name of the root POA is system-dependent and should not be relied upon by the application. In order to work properly with Portable Interceptors the name of the root POA must be the sequence containing only the string "RootPOA."

11.3.4.6 the_parent

readonly attribute POA the_parent;

This attribute identifies the parent of the POA. The parent of the root POA is null.

11.3.4.7 the_POAManager

readonly attribute POAManager the_POAManager;

This attribute identifies the POA manager associated with the POA.

11.3.4.8 activate_object

```
ObjectId activate_object(
    in Servant p_servant
) raises (ServantAlreadyActive, WrongPolicy);
```

This operation requires the **SYSTEM_ID** and **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the POA has the **UNIQUE_ID** policy and the specified servant is already in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object** operation generates an Object Id and enters the Object Id and the specified servant in the Active Object Map. The Object Id is returned.

11.3.4.9 activate_object_with_id

```
void activate_object_with_id(
    in              ObjectId oid,
    in Servant      p_servant
) raises (ObjectAlreadyActive, ServantAlreadyActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the CORBA object denoted by the Object Id value is already active in this POA (there is a servant bound to it in the Active Object Map), the ObjectAlreadyActive exception is raised. If the POA has the **UNIQUE_ID** policy and the servant is already in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object_with_id** operation enters an association between the specified Object Id and the specified servant in the Active Object Map.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this **POA**, the **activate_object_with_id** operation may raise the BAD_PARAM system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke **activate_object_with_id** on a **POA** that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**. A **POA** is not required to raise the BAD_PARAM exception in this case because, in the general case, accurate rejection of an invalid Object Id requires unbounded persistent memory of all previously generated Id values.

11.3.4.10 deactivate_object

```
void deactivate_object(
    in ObjectId oid
) raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

This operation causes the **ObjectId** specified in the **oid** parameter to be deactivated. An **ObjectId** that has been deactivated continues to process requests until there are no active requests for that **ObjectId**. Active requests are those requests that arrived before **deactivate_object** was called. A deactivated **ObjectId** is removed from the Active Object Map when all requests executing for that **ObjectId** have completed. Once an **ObjectId** has been removed from the Active Object Map, it may then be reactivated through the usual mechanisms.

The operation does not wait for requests to complete and always returns immediately after deactivating the **ObjectId**.

11.3.4.11 create_reference

```
Object create_reference (
    in CORBA::RepositoryId intf
) raises (WrongPolicy);
```

This operation requires the **SYSTEM_ID** policy; if not present, the WrongPolicy exception is raised.

This operation creates an object reference that encapsulates a POA-generated Object Id value and the specified interface repository id. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the appropriate servant manager to be invoked, if one is available. The generated Object Id value may be obtained by invoking **POA::reference_to_id** with the created reference.

11.3.4.12 create_reference_with_id

```
Object create_reference_with_id (
    in ObjectId              oid,
    in CORBA::RepositoryId   intf
);
```

This operation creates an object reference that encapsulates the specified Object Id and interface repository Id values. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this POA, the **create_reference_with_id** operation may raise the BAD_PARAM system exception with standard minor code 14. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke this operation on a POA that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**.

11.3.4.13 servant_to_id

```
ObjectId servant_to_id(
    in Servant              p_servant
) raises (ServantNotActive, WrongPolicy);
```

This operation requires a combination of the **RETAIN** policy and the **UNIQUE_ID** policies if invoked outside the context of an operation dispatched by this POA. If this operation is not invoked in the context of executing a request on the specified servant and the policies specified previously are not present, the WrongPolicy exception is raised.

This operation has two possible behaviors.

1. If the **POA** has both the **RETAIN** and the **UNIQUE_ID** policy and the specified servant is active, the Object Id associated with that servant is returned.
2. Otherwise, the ServantNotActive exception is raised.

11.3.4.14 servant_to_reference

```
Object servant_to_reference (
    in Servant              p_servant
) raises (ServantNotActive, WrongPolicy);
```

This operation requires the **RETAIN** and the **UNIQUE_ID** policies if invoked outside the context of an operation dispatched by this POA. If this operation is not invoked in the context of executing a request on the specified servant and the policies specified previously are not present, the WrongPolicy exception is raised.

This operation has two possible behaviors.

1. If the POA has both the **RETAIN** and the **UNIQUE_ID** policy and the specified servant is active, an object reference encapsulating the information used to activate the servant is returned.
2. Otherwise, the ServantNotActive exception is raised.

11.3.4.15 reference_to_servant

```
Servant reference_to_servant (
    in Object               reference
) raises (ObjectNotActive, WrongAdapter, WrongPolicy);
```

Table 11.1 summarizes the behavior of this operation based on the Object in question:

Table 11.1- Summary of reference_to_servant operation behavior

Object	Action
In AOM	Return Servant from AOM
Not in AOM	Raise ObjectNotActive

If the object reference was not created by this **POA**, the WrongAdapter exception is raised.

11.3.4.16 reference_to_id

```
ObjectId reference_to_id(
    in Object                    reference
) raises (WrongAdapter, WrongPolicy);
```

The WrongPolicy exception is declared to allow future extensions.

This operation returns the Object Id value encapsulated by the specified **reference**. This operation is valid only if the reference was created by the POA on which the operation is being performed. If the reference was not created by that POA, a WrongAdapter exception is raised. The object denoted by the reference does not have to be active for this operation to succeed.

11.3.4.17 id_to_servant

```
Servant id_to_servant(
    in ObjectId                  oid
) raises (ObjectNotActive, WrongPolicy);
```

If the specified **ObjectId** is in the Active Object Map, this operation returns the servant associated with that object in the Active Object Map. Otherwise the ObjectNotActive exception is raised.

11.3.4.18 id_to_reference

```
Object id_to_reference(
    in ObjectId                  oid
) raises (ObjectNotActive, WrongPolicy);
```

If an object with the specified Object Id value is currently active, a reference encapsulating the information used to activate the object is returned. If the Object Id value is not active in the POA, an ObjectNotActive exception is raised.

11.3.4.19 id

```
readonly attribute CORBA::OctetSeq id;
```

This returns the unique id of the POA in the process in which it is created.

This id is guaranteed unique for the life span of the POA in the process. For persistent POAs, this means that if a POA is created in the same path with the same name as another POA, these POAs are identical and, therefore, have the same id. For transient POAs, each POA is unique.

11.3.5 Current Operations

The **PortableServer::Current** interface, derived from **CORBA::Current**, provides method implementations with access to the identity of the object on which the method was invoked. The **Current** interface is provided to support servants that implement multiple objects, but can be used within the context of POA-dispatched method invocations on any servant. To provide location transparency, ORBs are required to support use of **Current** in the context of both locally and remotely invoked operations.

An instance of **Current** can be obtained by the application by issuing the **CORBA::ORB::resolve_initial_references("POACurrent")** operation. Thereafter, it can be used within the context of a method dispatched by the **POA** to obtain the **POA** and **ObjectId** that identify the object on which that operation was invoked.

PortableServer:Current is a local interface.

11.3.5.1 get_POA

```
POA get_POA()
    raises (NoContext);
```

This operation returns a reference to the POA implementing the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

11.3.5.2 *get_object_id*

```
ObjectId get_object_id()
    raises (NoContext);
```

This operation returns the **ObjectId** identifying the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

11.3.5.3 get_reference

```
Object get_reference()
    raises(NoContext);
```

This operation returns a locally manufactured reference to the object in the context of which it is called. If called outside the context of a POA dispatched operation, a NoContext exception is raised.

Note – This reference is not guaranteed to be identical to the original reference the client used to make the invocation, and calling the **Object::is_equivalent** operation to compare the two references may not necessarily return true.

11.3.5.4 get_servant

```
Servant get_servant()
    raises(NoContext);
```

This operation returns a reference to the servant that hosts the object in whose context it is called. If called outside the context of a POA dispatched operation, a NoContext exception is raised.

11.4 Consolidated IDL for PortableServer Module

11.4.1 CORBA/e Compact Profile

```
// IDL
// File: PortableServer.idl
```

```
#ifndef _PORTABLE_SERVER_IDL_
#define _PORTABLE_SERVER_IDL_

import ::CORBA;
module PortableServer {
      typeprefix PortableServer “org.omg”;
      local interface POA;           // forward declaration
      typedef sequence<POA> POAList;

      native Servant;

      typedef CORBA::OctetSeq ObjectId;

#   if ! defined (CORBA_E_MICRO)
      const CORBA::PolicyType LIFESPAN_POLICY_ID = 17;
      const CORBA::PolicyType ID_UNIQUENESS_POLICY_ID = 18;
      const CORBA::PolicyType ID_ASSIGNMENT_POLICY_ID = 19;
#   endif

#   if ! defined (CORBA_E_MICRO)
      enum LifespanPolicyValue {
            TRANSIENT,
            PERSISTENT
      };

      local interface LifespanPolicy : CORBA::Policy {
            readonly attribute LifespanPolicyValue value;
      };

      enum IdUniquenessPolicyValue {
            UNIQUE_ID,
            MULTIPLE_ID
      };

      local interface IdUniquenessPolicy : CORBA::Policy {
            readonly attribute IdUniquenessPolicyValue value;
      };

      enum IdAssignmentPolicyValue {
            USER_ID,
            SYSTEM_ID
      };

      local interface IdAssignmentPolicy : CORBA::Policy {
            readonly attribute IdAssignmentPolicyValue value;
      };

      // POAManager interface

      local interface POAManager {
            exception AdapterInactive{};
```

```
        enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

        void activate()
            raises(AdapterInactive);
        State get_state();
        string get_id();

    };

    // POA interface

    local interface POA {
        exception AdapterAlreadyExists {};
        exception AdapterNonExistent {};
        exception InvalidPolicy {unsigned short index;};
        exception NoServant {};
        exception ObjectAlreadyActive {};
        exception ObjectNotActive {};
        exception ServantAlreadyActive {};
        exception ServantNotActive {};
        exception WrongAdapter {};
        exception WrongPolicy {};

        // POA creation and destruction

#if ! defined (CORBA_E_MICRO)
        POA create_POA(  in string      adapter_name,
                         in POAManager   a_POAManager,
                         in CORBA::PolicyList policies)
            raises (AdapterAlreadyExists, InvalidPolicy);

        POA find_POA(    in string      adapter_name,
                         in boolean     activate_it)
            raises (AdapterNonExistent);
#endif

        void destroy(
            in boolean etherealize_objects,
            in boolean wait_for_completion);

#if ! defined (CORBA_E_MICRO)

        // POA attributes

        readonly attribute string the_name;
        readonly attribute POA the_parent;
        readonly attribute POAManager the_POAManager;

        // object activation and deactivation

        ObjectId activate_object(
            in Servant p_servant)
        raises (ServantAlreadyActive, WrongPolicy);

        void deactivate_object(
            in ObjectId oid)
        raises (ObjectNotActive, WrongPolicy);

        // reference creation operations

        Object create_reference (
            in CORBA::RepositoryId intf)
        raises (WrongPolicy);

        Object create_reference_with_id (
            in ObjectId oid,
            in CORBA::RepositoryId intf
        );

        // Identity mapping operations:

        ObjectId servant_to_id(
            in Servant p_servant)
        raises (ServantNotActive, WrongPolicy);

        Object servant_to_reference(
            in Servant p_servant)
        raises (ServantNotActive, WrongPolicy);

        Servant reference_to_servant(
            in Object reference)
        raises(ObjectNotActive, WrongAdapter, WrongPolicy);

        ObjectId reference_to_id(
            in Object reference)
        raises (WrongAdapter, WrongPolicy);

        Servant id_to_servant(
            in ObjectId oid)
        raises (ObjectNotActive, WrongPolicy);

        Object id_to_reference(in ObjectId oid)
            raises (ObjectNotActive, WrongPolicy);

        readonly attribute CORBA::OctetSeq id;
    };

    // Current interface

    local interface Current : CORBA::Current {
        exception NoContext { };
        POA get_POA()
```

```
        raises (NoContext);

        ObjectId get_object_id()
            raises (NoContext);

        Object get_reference()
            raises(NoContext);

        Servant get_servant()
            raises(NoContext);
    };
};
```

11.4.2 CORBA/e Micro Profile

```
// IDL
// File: PortableServer.idl
#ifndef _PORTABLE_SERVER_IDL_
#define _PORTABLE_SERVER_IDL_

import ::CORBA;
module PortableServer {
    typeprefix PortableServer “org.omg”;
    local interface POA;        // forward declaration
    typedef sequence<POA> POAList;

    native Servant;

    typedef CORBA::OctetSeq ObjectId;

    // POAManager interface

    local interface POAManager {
        exception AdapterInactive{};

        enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

        void activate()
            raises(AdapterInactive);
        State get_state();
        string get_id();

    };

    // POA interface
    local interface POA {
        exception ObjectNotActive {};
        exception ServantAlreadyActive {};
        exception ServantNotActive {};
        exception WrongAdapter {};
        exception WrongPolicy {};

        void destroy(
            in boolean etherealize_objects,
            in boolean wait_for_completion);
        // Factories for Policy objects

        LifespanPolicy
            create_lifespan_policy(in LifespanPolicyValue value);
        IdUniquenessPolicy  create_id_uniqueness_policy(
            in IdUniquenessPolicyValue value);
        IdAssignmentPolicy  create_id_assignment_policy(
            in IdAssignmentPolicyValue value);

        // POA attributes

        readonly attribute string the_name;
        readonly attribute POA the_parent;
        readonly attribute POAManager the_POAManager;

        // object activation and deactivation

        ObjectId activate_object(
            in Servant p_servant)
        raises (ServantAlreadyActive, WrongPolicy);

        void activate_object_with_id(in ObjectId    id,
                    in Servant     p_servant)
            raises (ServantAlreadyActive,
                    ObjectAlreadyActive,
                    WrongPolicy);

        void deactivate_object(
            in ObjectId oid)
        raises (ObjectNotActive, WrongPolicy);

        // reference creation operations

        Object create_reference (
            in CORBA::RepositoryId intf)
        raises (WrongPolicy);

        Object create_reference_with_id (
            in ObjectId oid,
            in CORBA::RepositoryId intf
        );

        // Identity mapping operations:

        ObjectId servant_to_id(
            in Servant p_servant)
```

```
        raises (ServantNotActive, WrongPolicy);

        Object servant_to_reference(
            in Servant p_servant)
        raises (ServantNotActive, WrongPolicy);

        Servant reference_to_servant(
            in Object reference)
        raises(ObjectNotActive, WrongAdapter, WrongPolicy);

        ObjectId reference_to_id(
            in Object reference)
        raises (WrongAdapter, WrongPolicy);

        Servant id_to_servant(
            in ObjectId oid)
        raises (ObjectNotActive, WrongPolicy);

        Object id_to_reference(in ObjectId oid)
            raises (ObjectNotActive, WrongPolicy);

        readonly attribute CORBA::OctetSeq id;
    };

    // Current interface

    local interface Current : CORBA::Current {
        exception NoContext { };

        POA get_POA()
        raises (NoContext);

        ObjectId get_object_id()
            raises (NoContext);

        Object get_reference()
            raises(NoContext);

        Servant get_servant()
            raises(NoContext);
    };
};
```

12 Real-time

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. Conformant implementations of the CORBA/*e* Micro Profile must comply with clauses 12.1.2.4 and 12.8 of this chapter.

12.1 Real-time Architecture

12.1.1 Goals of the Specification

In any architecture, there is a tension between a general purpose solution and supporting specialist applications. Real-time developers have to pay strict attention to the allocation of resources and to the predictability of system execution. By providing the developer with handles on managing resources and on predictability, CORBA/*e* sacrifices some of the general purpose nature of CORBA in order support the development of real-time systems.

Real-time development has further specialist areas: “hard” real-time and “soft” real-time; different resource contention protocols and scheduling algorithms, etc. This specification provides a set of real-time CORBA profiles that is sufficiently general to span these variations. The one restriction imposed by the specification is fixed priority scheduling. CORBA/*e* does not currently address dynamic scheduling.

The goals of the specification are to support developers in meeting real-time requirements by facilitating the end-to-end predictability of activities in the system and by providing support for the management of resources.

CORBA/*e*’s features brings to real-time system development the same benefits of implementation flexibility, portability, and interoperability that CORBA brought to client-server development.

There is one important non-goal for this specification. It is not a goal to provide a portability layer for the real-time operating system (RTOS) itself. The POSIX Real-time extensions already address this need. Real-time CORBA is compatible with the POSIX Real-time Extensions but by not wrapping the RTOS the specification facilitates the use of Real-time CORBA on operating systems that fall outside of the POSIX Real-time Extensions.

12.1.2 Approach to Real-time CORBA

12.1.2.1 The Nature of Real-time

There is a class of problems where some of the requirements relate the functionality of the system to Real-World time, be it measured in minutes or in microseconds. For these systems, timeliness is as important as functionality.

A parcel delivery service that commits to next day delivery across the country is relating the functional requirement of transporting a parcel from “A” to “B” to Real-world time; that is, “one day.” For the organization to meet this non-functional requirement, it must analyze the system, identify the activities, and bound the time taken to perform them. It must also decide what resources (people, planes, etc.) are allocated to the problem. the use of those resources in performing particular activities must be coordinated so that one activity doesn’t prejudice the Real-World time requirement of another activity. If the arrival rate of parcels and the isolation of resources from the outside world are known, then the organization can (ignoring component failures) guarantee “next day” delivery. If the arrival pattern of parcels is variable and the peak rate would suggest a large amount of resources (which would at other times be largely idle), then the organization could fall back to statistical predictability: offering “next day delivery or your money back.”

Relating functional requirements to real-world time may take several forms. A response time requirement might say that the occurrence of event “A” shall cause an event “B” within 24 hours. A throughput requirement might say that the system shall cope with 1000 occurrences of an event per hour. A statistical requirement might say that 95% of the occurrences of event “A” shall cause an event “B” within 24 hours. All these forms of requirement are real-time requirements. A system that meets real-time requirements is a real-time system.

12.1.2.2 Meeting Real-time Requirements

Deterministic behavior of the components of a real-time system promotes the predictability of the overall system. In order to decide *a priori* if a real-time requirement is met, the system must behave predictably. This can only happen if all the parts of the system behave deterministically and if they “combine” predictably.

The real-time interfaces and mechanisms provided by CORBA/*e* facilitate a predictable combination of the ORB and the application. The application manages the resources by using the real-time CORBA/*e* interfaces and the ORB’s mechanisms coordinate the activities that comprise the application. The real-time ORB relies upon the RTOS to schedule threads that represent activities being processed and to provide mutexes to handle resource contention.

12.1.2.3 Distributable Thread

This specification provides an abstraction for distributed real-time programming as an end-to-end schedulable entity termed *distributable thread*. Such a distributed execution model is essential to support:

- the statically-scheduled Real-time CORBA Base Architecture described in this and the following chapter; and
- as a foundation for dynamically scheduled real-time CORBA systems described in other Real-Time CORBA specifications.

This specification does not attempt to address more advanced issues such as fault tolerance, propagation of system information and control along the path of a distributable thread, etc. These facilities may be provided in a subsequent revision of this specification.

12.1.2.4 End-to-End Predictability

One goal of this specification is to provide a standard for CORBA ORB implementations that support end-to-end predictability. For the purposes of this specification, “end-to-end predictability” of timeliness in a fixed priority CORBA system is defined to mean:

- respecting thread priorities between client and server for resolving resource contention during the processing of CORBA invocations;
- bounding the duration of thread priority inversions during end-to-end processing;
- bounding the latencies of operation invocations.

A real-time CORBA/*e* system will include the following four major components, each of which must be designed and implemented in such a way as to support end-to-end predictability, if end-to-end predictability is to be achieved in the system as a whole:

1. the scheduling mechanisms in the OS;
2. the real-time ORB;
3. the communication transport;

4. the application(s).

Real-time ORBs conformant to this specification are still reliant on the characteristics of the underlying operating system and on the application if the overall system is to exhibit end-to-end predictability.

Note – An OS that implements the IEEE POSIX 1003.1-1996 Real-time Extensions has the necessary features to facilitate end-to-end predictability. It is still possible for an OS that doesn't implement some or all of the POSIX Real-time Extensions specification to support end-to-end predictability. Real-time CORBA is not restricted to such OSs.

12.1.3 Compatibility

12.1.3.1 Interoperability

CORBA/*e* does not prescribe an RT-IOP as an ESIOP. There are a number of pragmatic reasons for this. There are many specialized scenarios in which CORBA/*e* can be deployed. These different scenarios do not exhibit enough common characteristics to allow a common interaction protocol to be defined. Secondly, each scenario will impose a different transport protocol. Without agreeing on a common transport, interoperability isn't possible.

Instead of specifying an RT-IOP, this specification uses the "standard extension" mechanisms provided by IIOP. These mechanisms are GIOP ServiceContexts, IOR Profiles, and IOR Tagged Components. Using these it is possible for IIOP to provide protocol support for the mechanisms prescribed in CORBA/*e*.

The benefit is that two CORBA/*e* implementations will interoperate. Interoperability may not be as important for a real-time CORBA/*e* system as for a CORBA/*i* system because real-time dictates a measure of system-wide design control to deliver predictability and therefore also some control over which ORB to deploy.

The second benefit is that the specified extensions define what features of a vendors own Real-time IOP can be mapped onto IIOP. This allows vendors to bridge between different Real-time CORBA implementations.

12.1.3.2 Portability

Providing real-time applications with portability across real-time ORBs is a goal of this specification, providing a portability layer for real-time operating systems is not a goal. Basing such an RTOS wrapper on say, POSIX Real-time Extension would constrain the range of operating systems to which Real-time CORBA can add value.

Any real-time system will be carefully configured to meet its real-time requirements. This includes taking account of the behavior and timings of the ORB itself. Porting an application to a different Real-time ORB will necessitate that the application be reconfigured. Portability cannot be "write once run everywhere" for real-time CORBA/*e*. What it does do is reduce the risk to a development of having to port.

12.1.3.3 CORBA - Real-time CORBA Interworking

In many systems, real-time CORBA/*e* components will have to interwork with CORBA components. The interfaces (in particular IIOP extensions) are specified so that this is functionally possible. Clearly, in any given system, there will be timing and predictability implications that need to be considered if the real-time component is not to be compromised.

CORBA applications can be ported to Real-time ORBs. They simply will not make use of the extra functions provided. Porting a real-time application to a non-real-time ORB will sacrifice the predictability of that application but the two platforms are functionally equivalent.

12.1.4 Real-time CORBA/e Architectural Overview

Real-time CORBA defines a set of extensions to CORBA. Figure 12.1 shows the key real-time CORBA/*e* entities that are specified. The features that these relate to are described in overview in the following sections.

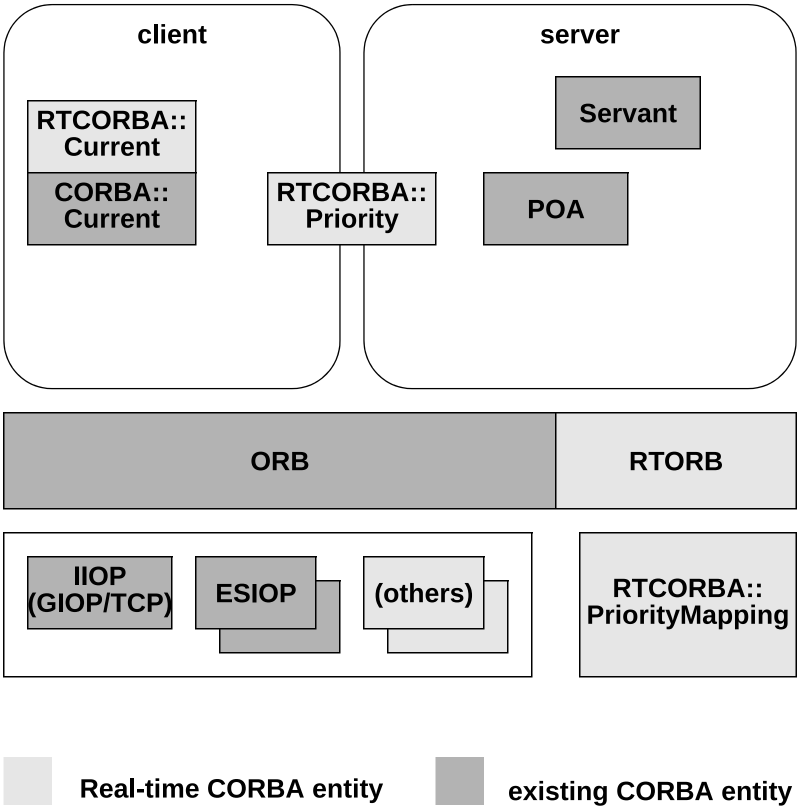

Figure 12.1- Real-time CORBA Extensions

12.1.4.1 Real-time CORBA Modules

All CORBA IDL specified by real-time extensions of CORBA/*e* is contained in new modules RTCORBA and RTPortableServer (with the exception of new service contexts, which are additions to the IOP module).

12.1.4.2 Thread Scheduling

The real-time extensions of CORBA/*e* uses threads as a schedulable entity. Generally, a thread represents a sequence of control flow within a single node. Threads for part of an activity. Activities are "scheduled" by coordination of the scheduling of their constituent threads. Real-time CORBA/*e* specifies interfaces through which the characteristics of a thread that are of interest can be manipulated. This interface is the Real-time CORBA Current interface.

Note – The Real-time CORBA view of a thread is compatible with the POSIX definition of a thread.

12.1.4.3 Real-time CORBA Priority

Real-time CORBA defines a universal, platform independent priority scheme called *Real-time CORBA Priority.* It is introduced to overcome the heterogeneity of different Operating System provided priority schemes, and allows Real-time CORBA applications to make prioritized CORBA invocations in a consistent fashion between nodes with different priority schemes.

For consistency, Real-time CORBA applications always should use CORBA Priority to express the priorities in the system, even if all nodes in a system use the same native thread priority scheme.

12.1.4.4 Native Priority and PriorityMappings

Real-time CORBA defines a **NativePriority** type to represent the priority scheme that is 'native' to a particular Operating System.

Priority values specified in terms of the Real-time CORBA Priority scheme must be mapped into the native priority scheme of a given scheduler before they can be applied to the underlying schedulable entities. On occasion, it is necessary for the reverse mapping to be performed, to obtain a Real-time CORBA Priority to represent the present native priority of a thread. The latter can occur, for example, when priority inheritance is in use, or when wishing to introduce an already running thread into a Real-time CORBA system at its present (native) priority.

To allow the Real-time ORB and applications to do both of these things, Real-time CORBA defines a **PriorityMapping** interface.

12.1.4.5 Real-time CORBA Current

Real-time CORBA defines a Real-time CORBA **Current** interface to provide access to the CORBA priority of a thread.

12.1.4.6 Priority Models

One goal of Real-time CORBA is to bound and to minimize priority inversion in CORBA invocations. One mechanism that is employed to achieve this is propagation of the activity priority from the client to the server, with the requirement that the server side ORB make the up-call at this priority (subject to any priority inheritance protocols that are in use).

CORBA/*e* supports a specific model for the priority at which a server handles requests from clients:

- Client Propagated Priority Model: in which the server honors the priority of the invocation, set by the client. The invocation's Real-time CORBA Priority is propagated to the server ORB and the server-side ORB maps this Real-time CORBA Priority into its own native priority scheme using its **PriorityMapping**.

Requests from non-Real-time CORBA ORBs; that is, ORBs that do not propagate a Real-time CORBA Priority with the invocation are handled at a priority specified by the server.

12.1.4.7 Real-time CORBA Mutexes and Priority Inheritance

The **Mutex** interface provides the mechanism for coordinating contention for system resources. Real-time CORBA/*e* specifies an **RTCORBA::Mutex** locality constrained interface, so that applications can use the same mutex implementation as the ORB.

A conforming CORBA/*e* implementation must provide an implementation of **Mutex** that implements some form of priority inheritance protocol. This may include, but is not limited to, simple priority inheritance or a form of priority ceiling protocol. The mutexes that Real-time CORBA makes available to the application must have the same priority inheritance properties as those used by the ORB to protect resources. This allows a consistent priority inheritance scheme to be delivered across the whole system.

12.1.4.8 Priority Banded Connections

To reduce priority inversion due to use of a non-priority respecting transport protocol, RT CORBA provides the facility for a client to communicate with a server via multiple connections, with each connection handling invocations that are made at a different CORBA priority or range of CORBA priorities. The selection of the appropriate connection is transparent to the application, which uses a single object reference as normal.

12.1.4.9 Invocation Timeouts

Real-time CORBA applications may set a timeout on an invocation in order to bound the time that the client application is blocked waiting for a reply. This can be used to improve the predictability of the system.

12.2 Real-time ORB

CORBA/*e* defines an extension to the **CORBA::ORB** interface, **RTCORBA::RTORB**. This interface is not derived from **CORBA::ORB** as the latter is expressed in pseudo IDL, for which inheritance is not defined. Nevertheless, **RTORB** is conceptually the extension of the ORB interface.

The **RTORB** interface provides operations to create and destroy other constituents of a Real-time ORB.

There is a single instance of **RTCORBA::RTORB** per instance of **CORBA::ORB**. The object reference for the **RTORB** is obtained by calling **ORB::resolve_initial_references** with the **ObjectId** "**RTORB**".

RTCORBA::RTORB is a local interface. The reference to the **RTORB** object may not be passed as a parameter of an IDL operation nor may it be stringified. Any attempt to do so shall result in a MARSHAL system exception (with a Standard Minor Exception Code of 4).

```
// IDL
module RTCORBA {

    local interface RTORB {

        ...

    };

};
```

12.2.1 Real-time ORB Initialization

Real-time ORB initialization occurs within the **CORBA::ORB_init** operation. That is a Real-time ORB's implementation of **CORBA::ORB_init** shall perform any actions necessary to initialize the real-time capability of the ORB.

In order to give the developer some control over a Real-time ORB's use of priorities the **ORB_init** operation shall be capable of handling an argv element of the form:

–ORBRTpriorityrange<optional-white-space><short>,<short>

Where **<short>** is a string encoding of an integer between 0 and 32767. The first integer should be smaller than the second. If the argv element string does not conform to these constraints, then a BAD_PARAM system exception shall be raised.

The two integers represent a range of CORBA Priorities available for use by ORB internal threads. Note that priority of Real-time CORBA application threads is controlled by other mechanisms. If the ORB cannot map these integers onto the native priority scheme, then it shall raise a DATA_CONVERSION system exception.

If the ORB deems the range of priorities to be too narrow for it to function properly, then it shall raise an INITIALIZE system exception (with a Standard Minor Exception Code of 1). For example, an implementation may not be able to function with less than, say, three distinct priorities without risking deadlock.

12.2.2 Real-time CORBA System Exceptions

CORBA/*e* provides a more constraining environment for an application than the environment provided by CORBA/*i*. This is reflected in the additional circumstances in which system exceptions can be generated. These circumstances need to be differentiated from the use of the same exception in CORBA.

Real-time CORBA/*e* uses many of the Standard System Exceptions with the same meaning as applies in CORBA. These uses need no differentiation. Where the use of a CORBA Standard System Exception has a meaning particular to real-time CORBA, Standard Minor Exception Codes are assigned.

Table 12.1- Standard Minor Exception Codes used for Real-time CORBA

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
MARSHAL	4	Attempt to marshal local object.
DATA_CONVERSION	2	Failure of PriorityMapping object.
INITIALIZE	1	Priority range too restricted for ORB.
BAD_INV_ORDER	18	Attempt to reassign priority.
NO_RESOURCES	2	No connection for request's priority.

12.3 Native Thread Priorities

A real-time operating system (RTOS) sufficient to use for implementing a Real-time ORB compliant with this specification, will have some discrete representation of a thread priority. This representation typically specifies a range of values and a direction for which values, higher or lower, represent the higher priority. The particular range and direction in this priority representation varies from RTOS to RTOS. This specification refers to the RTOS specific thread priority representation as a *native thread priority scheme*. The priority values of this scheme are referred to as *native thread priorities*.

Native thread priorities are used to designate the execution eligibility of threads. The ordering of native thread priorities is such that a thread with higher native priority is executed at the exclusion of any threads in the system with lower native priorities.

A native thread priority is an integer value that is the basis for resolving competing demands of threads for resources. Whenever threads compete for processors or ORB implementation-defined resources, the resources are allocated to the thread with the highest native thread priority value.

The *base native thread priority* of a thread is defined as the native priority with which it was created, or to which it was later set explicitly. The initial value of a thread's base native priority is dependent on the semantics of the specific operating environment. Hence it is implementation specific.

At all times, a thread also has an *active native thread priority*, which is the result of considering its base native thread priority together with any priorities it inherits from other sources; for example, threads or mutexes. An active native thread priority is set implicitly as a result of some other action. Its value is only temporary, at some point it will return to the base native thread priority.

Priority inheritance is the term used for the process by which the native thread priority of other threads is used in the evaluation of a thread's active native thread priority. A *priority inheritance protocol* must be used by a conforming Real-time CORBA ORB to implement the execution semantics of threads and mutexes. It is an implementation issue as to whether the Real-time ORB implements simple priority inheritance, immediate ceiling locking protocol, original ceiling locking protocol, or some other priority inheritance protocol.

Whichever priority inheritance protocol is used, the native thread priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists. At the point when a thread stops inheriting a native thread priority from another source, its active native thread priority is re-evaluated.

The thread's active native priority is used when the thread competes for processors. Similarly, the thread's active native priority is used to determine the thread's position in any queue; that is, dequeuing occurs in native thread priority order.

Native priorities have an IDL representation in Real-time CORBA, which is of type short:

```
// IDL
module RTCORBA {

    typedef short NativePriority;

};
```

This means that native priorities must be integer values in the range -32768 to +32767. However, for a particular RTOS, the valid range will be a sub-range of this range.

Real-time CORBA does not support the direct use of native priorities: instead, the application programmer uses CORBA Priorities, which are defined in the next section. However, applications will still use native priorities where they make direct use of RTOS features.

12.4 CORBA Priority

To overcome the heterogeneity of RTOSs, that is different RTOSs having different native thread priority schemes, Real-time CORBA defines a CORBA Priority that has a uniform representation system-wide. CORBA Priority is represented by the **RTCORBA::Priority type**:

```
//IDL
module RTCORBA {

   typedef short Priority;
```

```
    const Priority minPriority =     0;
    const Priority maxPriority = 32767;

};
```

A signed short is used in order to accommodate the Java language mapping. However, only values in the range 0 (minPriority) to 32767 (maxPriority) are valid. Numerically higher **RTCORBA::Priority** values are defined to be of higher priority.

For each RTOS in a system, CORBA priority is mapped to the native thread priority scheme. CORBA priority thus provides a common representation of priority across different RTOSs.

12.5 CORBA Priority Mappings

Real-time CORBA defines the concept of a **PriorityMapping** between CORBA priorities and native priorities. The concept is defined as an IDL native type so that the mechanism by which priorities are mapped is exposed to the user. Native is chosen rather than interface (even if locality constrained) because the full capability of the ORB; for example, POA policies and CORBA exceptions are too heavyweight for this use. Furthermore, a CORBA interface would entail the creation and registration of an object reference.

```
// IDL
module RTCORBA {

     native PriorityMapping;

};
```

Language mapping for this IDL native are defined for C, C++, Ada, and Java later in this section.

A Real-time ORB shall provide a default mapping for each platform; that is, RTOS that the ORB supports. Furthermore, a Real-time ORB shall provide a mechanism to allow users to override the default priority mapping with a priority mapping of their own.

The **PriorityMapping** is a programming language object rather than a CORBA Object and therefore the normal mechanism for coupling an implementation to the code that uses it (an object reference) doesn't apply. This specification does not prescribe a particular mechanism to achieve this coupling.

Note – Possible solutions include: recourse to build/link tools and provision of proprietary interfaces. Other solutions are not precluded.

12.5.1 C Language binding for PriorityMapping

```
/* C */
CORBA_boolean RTCORBA_PriorityMapping_to_native (
         RTCORBA_Priority            corba_priority,
         RTCORBA_NativePriority* native_priority );

CORBA_boolean RTCORBA_PriorityMapping_to_CORBA (
         RTCORBA_NativePriority native_priority,
         RTCORBA_Priority*        corba_priority );
```

12.5.2 C++ Language binding for PriorityMapping

```
// C++
namespace RTCORBA {

   class PriorityMapping {
         public:
               virtual CORBA::Boolean to_native (
                           RTCORBA::Priority corba_priority,
                           RTCORBA::NativePriority &native_priority );
               virtual CORBA::Boolean to_CORBA (
                           RTCORBA::NativePriority native_priority,
                           RTCORBA::Priority &corba_priority );
   };
};
```

12.5.3 Ada Language binding for PriorityMapping

```
-- Ada
package RTCORBA.PriorityMapping is

   type Object is tagged private;

   procedure To_Native (
         Self                    : in Object ;
         CORBA_Priority          : in RTCORBA.Priority ;
         Native_Priority         : out RTCORBA.NativePriority ;
         Returns                 : out CORBA.Boolean ) ;

   procedure To_CORBA (
         Self                    : in Object ;
         Native_Priority         : in RTCORBA.NativePriority ;
         CORBA_Priority          : out RTCORBA.Priority ;
         Returns                 : out CORBA.Boolean ) ;

end RTCORBA.PriorityMapping ;
```

12.5.4 Java Language binding for PriorityMapping

```
// Java
package org.omg.RTCORBA;
   public class PriorityMapping {

         boolean to_native (
               short corba_priority,
               org.omg.CORBA.ShortHolder native_priority
```

```
        );
        boolean to_CORBA (
              short native_priority,
              org.omg.CORBA.ShortHolder corba_priority
        );
}
```

12.5.5 Semantics

The priority mappings between native priority and CORBA priority are defined by the implementations of the **to_native** and **to_CORBA** operations of a PriorityMapping object (note, not a CORBA Object). The **to_native** operation accepts a CORBA Priority value as an in parameter and maps it to a native priority, which is given back as an out parameter. Conversely, **to_CORBA** accepts a **NativePriority** value as an in parameter and maps it to a CORBA Priority value, which is again given back as an out parameter.

As the mappings are used by the ORB, and may be used more than once in the normal execution of an invocation, their implementations should be as efficient as possible. For this reason, the mapping operations may not raise any CORBA exceptions, including system exceptions. The ORB is not restricted from making calls to the **to_native** and/or **to_CORBA** operations from multiple threads simultaneously. Thus, the implementations should be re-entrant.

Rather than raising a CORBA exception upon failure, a boolean return value is used to indicate mapping failure or success. If the priority passed in can be mapped to a priority in the target priority scheme, TRUE is returned and the value is returned as the out parameter. If it cannot be mapped, FALSE is returned and the value of the out parameter is undefined.

Both **to_native** and **to_CORBA** must return FALSE when passed a priority that is outside of the valid priority range of the input priority scheme. For **to_native** this means when it is passed a short value outside of the CORBA Priority range, 0-32767; that is, a negative value. For **to_CORBA** this means when it is passed a short value outside of the native priority range used on that RTOS. This range will be platform specific.

Neither **to_native** nor **to_CORBA** is obliged to map all valid values of the input priority scheme (the CORBA Priority scheme or the native priority scheme, respectively). This allows mappings to be produced that do not use all values of the native priority scheme of a particular scheduler and/or that do not use all values of the CORBA Priority scheme.

When the ORB receives a FALSE return value from a mapping operation that is called as part of the processing of a CORBA invocation, processing of the invocation proceeds no further. A DATA_CONVERSION system exception (with a Standard Minor Exception Code of 2) is raised to the application making the invocation. Note that it may not be possible to raise an exception to the application if the failure occurs on a call to a mapping operation made on the server side of a oneway invocation.

A Real-time ORB cannot assume that the priority mapping is idempotent. Users should be aware that a mapping that produces different results for the same inputs will make the goal of a schedulable system harder to obtain. Users may choose to implement a priority mapping that changes (through other, user specified interfaces). Users should however note that post-initialization changes to the mapping may well compromise the ORB’s ability to deliver a consistently schedulable system.

12.6 Real-time Current

The **RTCORBA::Current** interface, derived from **CORBA::Current**, provides access to the CORBA Priority (and hence indirectly to the native priority also) of the current thread. The application can obtain an instance of Current by invoking the **CORBA::ORB::resolve_initial_references(“RTCurrent”)** operation.

A Real-time CORBA Priority may be associated with the current thread, by setting the priority attribute of the **RTCORBA::Current** object:

```
//IDL
module RTCORBA {

    local interface Current : CORBA::Current {
        attribute Priority base_priority;
    };

};
```

A BAD_PARAM system exception shall be thrown if an attempt is made to set the priority to a value outside the range 0 to 32767.

When the attribute is set to a valid Real-time CORBA Priority value, the value is immediately used to set the base native priority of the thread. The native priority value to use is determined by calling **PriorityMapping::to_native** on the installed PriorityMapping. The native thread priority shall be set before the set attribute call returns.

If the **to_native** call returns FALSE or if the returned native thread priority is illegal for the particular underlying RTOS, then a Real-time ORB shall raise a DATA_CONVERSION system exception (with a Standard Minor Exception Code of 2). In this case the priority attribute shall retain its value prior to the set attribute call.

Once a thread has a CORBA Priority value associated with it, the behavior when it makes an invocation upon a CORBA Object depends on the value of the **PriorityModelPolicy** of that target object.

Retrieving the value of this attribute returns the last value that was set from the current thread. If this attribute has not previously been set for the current thread, attempting to retrieve the value causes an INITIALIZE System Exception to be raised.

12.7 Real-time CORBA Priority Models

Note – CORBA/*e* supports the Client-Propagated Priority Model policy only.

12.7.1 Scope of PriorityModelPolicy

The **PriorityModelPolicy** is applied to a Real-time POA at the time of POA creation. This is either through an ORB level default having previously been set or by including it in the policies parameter to **create_POA**. An instance of the **PriorityModelPolicy** is created with the **create_priority_model_policy** operation. The attributes of the policy are initialized with the parameters of the same name.

```
//IDL
module RTCORBA {

    local interface RTORB {
        ...
        PriorityModelPolicy create_priority_model_policy (
                        in PriorityModel priority_model,
                        in Priority server_priority
        );
    };

};
```

The **PriorityModelPolicy** is a client-exposed policy; that is, propagated from the server to the client in IORs. It is propagated in a **PolicyValue** in a **TAG_POLICIES** Profile Component, as specified by the CORBA QoS Policy Framework.

When an instance of **PriorityModelPolicy** is propagated, the **PolicyValue**'s ptype has the value **PRIORITY_MODEL_POLICY_TYPE** and the pvalue is a CDR encapsulation containing an **RTCORBA::PriorityModel** and an **RTCORBA::Priority**.

Note – Client-exposed policies and the mechanism for their propagation are defined in the "Policies" chapter.

The **PriorityModelPolicy** is propagated so that the client ORB knows which Priority Model the target object is using. This allows it to determine whether to send the Real-time CORBA priority with invocations on that object, and allows it to select the band connection to invoke over based on the declared priority in the tagged component.

The client may not override the **PriorityModelPolicy**.

12.7.2 Client Propagated Priority Model

If the target object supports the **CLIENT_PROPAGATED** value of the **PriorityModelPolicy**, the CORBA Priority is carried with the CORBA invocation and is used to ensure that all threads subsequently executing on behalf of the invocation run at the appropriate priority. The propagated CORBA Priority becomes the CORBA Priority of any such threads. These threads run at a native priority mapped from that CORBA Priority. The CORBA Priority is also passed back from server to client, so that it can be used to control the processing of the reply by the client ORB.

The CORBA Priority is propagated from client to server, and back again, in a CORBA Priority service context, which is passed in the invocation request and reply messages.

```
module IOP {

    const ServiceId        RTCorbaPriority = 10;

};
```

The **context_data** contains the **RTCORBA::Priority** value as a CDR encapsulation of an IDL short type.

Note – The **RTCorbaPriority** const should be added to a future version of GIOP.

The thread that runs the servant code, initially has the CORBA Priority of the invoking thread. Therefore if, as part of the processing of this request it makes CORBA invocations to other objects, these onward invocations will be made with the same CORBA Priority. If the CORBA Priority of the thread running the servant code is changed by the application, any subsequent onward invocations will be made with this new priority.

Note that priorities may be changed implicitly, by the platform (RT ORB + RTOS) whilst the servant code is executing due to priority inheritance.

12.8 Mutex Interface

Real-time CORBA defines the following **Mutex** interface:

```
//IDL
module RTCORBA {

    local interface Mutex    {

        void lock( );
        void unlock( );
        boolean try_lock(in TimeBase::TimeT max_wait);
            // if max_wait = 0 then return immediately
    };

    local interface RTORB {

        ...
        Mutex create_mutex();
        void destroy_mutex( in Mutex the_mutex );
        ...
    };
};
```

A new **RTCORBA::Mutex** object is obtained using the **create_mutex()** operation of **RTCORBA::RTORB**.

A Mutex object has two states: locked and unlocked. Mutex objects are created in the unlocked state. When the Mutex object is in the unlocked state the first thread to call the **lock()** operation will cause the Mutex object to change to the locked state. Subsequent threads that call the **lock()** operation while the Mutex object is still in the locked state will block until the owner thread unlocks it by calling the **unlock()** operation.

Note – If a Real-time ORB is to run on a shared memory multi-processor, then the Mutex implementation must ensure that the lock operations are atomic to all processors.

The **try_lock()** operation works like the **lock()** operation except that if it does not get the lock within **max_wait** time it returns FALSE. If the **try_lock()** operation does get the lock within the **max_wait** time period, it returns TRUE.

The mutex returned by **create_mutex** must have the same priority inheritance properties as those used by the ORB to protect resources. If a Real-time CORBA implementation offers a choice of priority inheritance protocols, or offers a protocol that requires configuration, the selection or configuration will be controlled through an implementation specific interface.

While a thread executes in a region protected by a mutex object, it can be pre-empted only by threads whose active native thread priorities are higher than either the ceiling or inherited priority of the mutex object.

Note – The protocol implemented by the Mutex (which priority inheritance or priority ceiling protocol) is not prescribed. Real-time CORBA is intended for a wide range of RTOSs and the choice of protocol will often be predicated on what the RTOS does.

12.9 Implicit and Explicit Binding

Real-time CORBA makes use of the **CORBA::Object::validate_connection** operation to allow client applications to control when a binding is made on an object reference.

Note – validate_connection and the definition of binding that it uses are defined in the *Common Object Request Broker Architecture (CORBA)* specification, CORBA Messaging chapter.

Using **validate_connection** on a currently unbound object reference causes binding to occur. Real-time CORBA refers to the use of **validate_connection** to force a binding to be made as ‘explicit binding.’ If an object reference is not explicitly bound, binding will occur at an ORB specific time, which may be as late as the time of the first invocation upon that object reference. This is referred to as ‘implicit binding,’ and is the default CORBA behavior unless an explicit bind is performed.

Real-time applications may wish to use explicit binding to force any binding related overhead (including the passing of messages between the client and server) to be incurred ahead of the first invocation on an object reference. This can improve the performance and predictability of the first invocation, and hence the predictability of the application as a whole. The explicit bind may, for example, be performed during system initialization.

Once an explicit binding has been set up, via **validate_connection**, it is possible that the underlying transport connection (or other associated resources) may fail or may be reclaimed by the ORB. Rather than mandate that this shall not happen, it is left as a Quality of Implementation issue as to what guarantees of enduring availability an explicit binding provides.

The client-side applicable Real-time CORBA policies are applied to a binding in the same way as any other client-side applicable CORBA policies: using the **set_policy_overrides** operations at the ORB, Current, or Object scope (as defined in the CORBA QoS Policy Framework.)

The client-side applicable Real-time CORBA policies have the same effect whether they are applied to an implicit or explicit bind.

12.10 Priority Banded Connections

Priority banded connections are administered through the use of the Real-time CORBA **PriorityBandedConnectionsPolicy** Policy type:

```
// IDL
module RTCORBA {

    struct PriorityBand {
        Priority low;
        Priority high;
    };

    typedef sequence <PriorityBand> PriorityBands;

    // PriorityBandedConnectionPolicy
    const CORBA::PolicyType
        PRIORITY_BANDED_CONNECTION_POLICY_TYPE = 45;
    local interface PriorityBandedConnectionPolicy : CORBA::Policy {

        readonly attribute PriorityBands priority_bands;

    };

    local interface RTORB {
        ...
        PriorityBandedConnectionPolicy
                create_priority_banded_connection_policy (
                        in PriorityBands priority_bands
        );
    };

};
```

An instance of the **PriorityBandedConnectionPolicy** is created with the **create_priority_banded_connection_policy** operation. The attribute of the policy is initialized with the parameter of the same name.

The **PriorityBands** attribute of the policy may be assigned any number of **PriorityBands**. **PriorityBands** that cover a single priority (by having the same priority for their low and high values) may be mixed with those covering ranges of priorities. No priority may be covered more than once. The complete set of priorities covered by the bands do not have to form one contiguous range, nor do they have to cover all CORBA Priorities. If no bands are provided, then a single connection will be established.

Once the binding has been successfully made, an attempt to make an invocation with a Real-time CORBA Priority, which is not covered by one of the bands will fail. The ORB shall raise a NO_RESOURCES system exception (with a Standard Minor Exception Code of 2). Hence, a policy specifying only one band can be used to restrict a client's invocations to a range of priorities.

Note that the origin of the Real-time CORBA Priority value that is used to select which banded connection to use depends on the Priority Model of the target object. When invoking on an Object that is using the Client Propagated Priority Model, the client-set Real-time CORBA Priority is used to choose the band.

12.10.1 Scope of PriorityBandedConnectionPolicy

The **PriorityBandedConnectionPolicy** is applied on the client-side only, at the time of binding to a CORBA Object. However, the policy may be set from the client or server side. On the server, it may be applied at the time of POA creation, in which case the policy is client-exposed and is propagated from the server to the client in interoperable Object References. It is propagated in a **PolicyValue** in a TAG_POLICIES Profile Component, as specified by the CORBA QoS Policy Framework.

When an instance of **PriorityBandedConnectionPolicy** is propagated, the **PolicyValue**'s ptype has the value **PRIORITY_BANDED_CONNECTION_POLICY_TYPE** and the pvalue is a CDR encapsulation containing an **RTCORBA::PriorityBands** type, which is a sequence of instances of **RTCORBA::PriorityBand**. Each **RTCORBA::PriorityBand** is in turn represented by a pair of **RTCORBA::Priority** values, which represent the low and high values for that band.

If the **PriorityBandedConnectionPolicy** is set on both the server and client-side an attempt to bind will fail with an INV_POLICY system exception. The client application may use **validate_connection** to establish that this was the cause of binding failure and may set the value of its copy of the policy to an empty **PriorityBands** and attempt to rebind using just the configuration from the server-provided copy of the policy.

12.10.2 Binding of Priority Banded Connection

Whether bands are configured from the client or server-side, the banded connection is always initiated from the client-side.

In order to allow the server-side ORB to identify the priority band that each connection is associated with, information on that connection's band range is passed with first request on each banded connection. This is done by means of an **RTCorbaPriorityRange** service context:

```
// IDL
module IOP {

    const ServiceId RTCorbaPriorityRange = 11;

};
```

The **context_data** contains the CDR encapsulation of two **RTCORBA::Priority** values (two short types.) The first indicates the lowest priority and the second the highest priority in the priority band handled by the connection.

Once a priority band has been associated with a connection it cannot be reconfigured during the life-time of the connection. If an ORB receives a second, or subsequent, **RTCorbaPriorityRange** service context containing a different priority band definition, then it shall raise a BAD_INV_ORDER system exception (with a Standard Minor Exception Code of 18). If the priority band is the same as the connection's configuration, then processing shall proceed.

In case of an explicit bind (via **validate_connection**), this service context is passed on a request message for a '**_bind_priority_band**' implicit operation. This implicit operation is defined for Real-time CORBA only at this time. It is possible that non-Real-time ORB might receive such a request message. If so it is anticipated (but not prescribed) that it will reply with a BAD_OPERATION system exception with standard minor code 2. A future version of IIOP should formalize Real-time CORBA's use of the '**_bind_priority_band**' operation name in a GIOP Request message. Note that there is no API exposed for this implicit operation (unlike, for example, '**_is_a**').

When sending a '**_bind_priority_band**' request, a Real-time ORB shall marshall no parameters and the object key of the object being bound to shall be used as the request 'target.' The request shall be handled by the ORB, no servant implementation of this implicit operation will be invoked.

When a Real-time-ORB receives a **_bind_priority_band** Request it should allocate resources to the connection and configure those resources appropriately to the priority band indicated in the **ServiceContext**. Having done this the ORB shall send a "SUCCESS" Reply message. If the priority band passed is not well-formed; that is, it contains a negative number or the first value is higher than the second, then the ORB shall raise a BAD_PARAM system exception. If either of the priorities cannot be mapped onto native thread priorities; that is, to-native returns FALSE, then the ORB shall raise a DATA_CONVERSION system exception (with a Standard Minor Exception Code of 2). If the priority band is inconsistent with the ORB's priority configuration, then the ORB shall raise an INV_POLICY system exception. If the server-side ORB cannot configure resources to support a well-formed band specification, then a NO_RESOURCES exception shall be returned.

A **_bind_priority_band** request message is sent on the connection for each band and must complete successfully; that is, yield a SUCCESS Reply message for all connections, before **validate_connection** returns success. If any one **_bind_priority_band** fails, then the entire banded connection binding fails. In this way, **validate_connection** sets up all the banded connections at time of binding.

If the service context is omitted on a **_bind_priority_band** request message, then the ORB shall raise a BAD_PARAM system exception.

A **bind_priority_band** is not performed in the case of an implicit bind, as it occurs at a time when a request is about to be sent on the connection representing the priority band that covers the current invocation priority. There is no point delaying the application request. Instead, the '**RTCorbaPriorityRange**' service context is passed on this first invocation request.

Thus, the implicit binding of a banded connection has the behavior that each band connection is only set up the first time an invocation is made from the client with an invocation priority in that band. This behavior offers consistency: the first invocation made on each band incurs any latency and predictability cost associated with binding. If no invocations are ever made in the priority range of a given band, its connection will never be established.

12.11 Invocation Timeout

Real-time CORBA uses the existing CORBA timeout policy, **Messaging::RelativeRoundtripTimeoutPolicy**, to allow a timeout to be set for the receipt of a reply to an invocation. The policy is used where it is set, to set a timeout in the client ORB. If a timeout expires, the server is not informed. Real-time CORBA does not require the policy to be propagated with the invocation, which the **RelativeRoundtripTimeoutPolicy** specification allows in support of message routers.

Note – The **RelativeRoundtripTimeoutPolicy** is specified in Section 7.4.3.4, "interface RelativeRoundtripTimeoutPolicy," on page 17.

12.12 Consolidated IDL

```
// IDL
module IOP {
    const ServiceId        RTCorbaPriority = 10;
    const ServiceId        RTCorbaPriorityRange = 11;
};

//File: RTCORBA.idl
#ifndef _RT_CORBA_IDL_
#define _RT_CORBA_IDL_
#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"

#include <orb.idl>
#include <iop.idl>
#include <TimeBase.idl>
#else
import ::CORBA;
import ::IOP;
import ::TimeBase;
#endif

// IDL
module RTCORBA {
    typedef short NativePriority;

    typedef short Priority;

    const Priority minPriority = 0;
    const Priority maxPriority = 32767;

    native PriorityMapping;

    // Priority Model Policy
    const CORBA::PolicyType PRIORITY_MODEL_POLICY_TYPE = 40;

    struct PriorityBand {
        Priority low;
        Priority high;
    };

    typedef sequence <PriorityBand> PriorityBands;

    // PriorityBandedConnectionPolicy
    const CORBA::PolicyType
        PRIORITY_BANDED_CONNECTION_POLICY_TYPE = 45;

    local interface PriorityBandedConnectionPolicy : CORBA::Policy {
        readonly attribute PriorityBands priority_bands;
    };

    local interface Current : CORBA::Current {
        attribute Priority the_priority;
    };

    local interface Mutex    {

        void lock( );
        void unlock( );
        boolean try_lock ( in TimeBase::TimeT max_wait );
        // if max_wait = 0 then return immediately
    };

    local interface RTORB {

        Mutex create_mutex( );
        void destroy_mutex( in Mutex the_mutex );

        PriorityModelPolicy create_priority_model_policy (
                 in PriorityModel priority_model,
                in Priority      server_priority
                );
          PriorityBandedConnectionPolicy
                      create_priority_banded_connection_policy (
                                  in PriorityBands priority_bands
          );

     }; // End interface RTORB

}; // End module RTCORBA
#endif // _RT_CORBA_IDL_
```

13 Naming Service

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. There are no requirements for conformant implementations of the CORBA/*e* Micro Profile in this chapter.

13.1 Overview

The **CosNaming** module is a collection of interfaces that together define the Naming Service. This module contains three interfaces:

- The **NamingContext** interface
- The **BindingIterator** interface
- The **NamingContextExt** interface

This section describes these interfaces and their operations in detail.

The **CosNaming** module is shown below.

Note – Istring was a “placeholder for a future IDL internationalized string data type” in the original specification. It is maintained solely for compatibility reasons.

13.1.1 Resolution of Compound Names

In this specification operations that are performed on compound names recursively perform the *equivalent* of a **resolve** operation on all but the last component of a name before performing the operation on the final name component. The general form is defined as follows:

ctx->op(<c1; c2; ...; cn>) equiv

ctx->resolve(<c1>)->resolve(<c2; cn-1>)->op(<cn>)

where ctx is a naming context, <c1; ...; cn> a compound name, and op a naming context operation.

Note – The intermediate components, <c1: ...; cn> of the compound name must have been bound using **bind_context** or **rebind_context** to take part in the resolve.

13.2 NamingContext Interface

The following sections describe the naming context data types and interface in detail.

13.2.1 Structures

13.2.1.1 NameComponent

```
struct NameComponent {
    Istring Id;
    Istring kind;
};
```

A name component consists of two attributes: the identifier attribute - **id** and the kind attribute - **kind**.

Both of these attributes are arbitrary-length strings of ISO Latin-1 characters, excluding the ASCII **NUL** character.

When comparing two **NameComponents** for equality both the **id** and the **kind** field must match in order for two **NameComponents** to be considered identical. This applies for zero-length (empty) fields as well. Name comparisons are case sensitive.

An implementation may place limitations on the characters that may be contained in a name component, as well as the length of a name component. For example, an implementation may disallow certain characters, may not accept the empty string as a legal name component, or may limit name components to some maximum length.

13.2.1.2 Name

A name is a sequence of **NameComponent**s. The empty sequence is not a legal name. An implementation may limit the length of the sequence to some maximum. When comparing **Name**s for equality, each **NameComponent** in the first name must match the corresponding **NameComponent** in the second **Name** for the names to be considered identical.

13.2.1.3 Binding

```
enum BindingType { nobject, ncontext };
struct Binding {
    Name            binding_name;
    BindingType     binding_type;
};
typedef sequence<Binding> BindingList;
```

This type is used by the **NamingContext::list**, **BindingIterator::next_n**, and **BindingIterator::next_one** operations. A **Binding** contains a **Name** in the member **binding_name**, together with the **BindingType** of that **Name** in the member **binding_type**.

Note – The **binding_name** member is incorrectly typed as a **Name** instead of a **NameComponent**. For compatibility with the original **CosNaming** specification this incorrect definition has been retained. The **binding_name** is used as a **NameComponent** and will always be a **Name** with length of 1.

The value of **binding_type** is **ncontext** if a **Name** denotes a binding created with one of the following operations:

- **bind_context**
- **rebind_context**
- **bind_new_context**

For bindings created with any other operation, the value of **BindingType** is **nobject**.

13.2.2 Exceptions

The Naming Service exceptions are defined below.

13.2.2.1 NotFound

```
exception NotFound {
    NotFoundReason why;
    Name rest_of_name;
};
```

This exception is raised by operations when a component of a name does not identify a binding or the type of the binding is incorrect for the operation being performed. The **why** member explains the reason for the exception and the **rest_of_name** member contains the remainder of the non-working name:

- **missing_node**

  The first name component in **rest_of_name** denotes a binding that is not bound under that name within its parent context.

- **not_context**

  The first name component in **rest_of_name** denotes a binding with a type of **nobject** when the type **ncontext** was required.

- **not_object**

  The first name component in **rest_of_name** denotes a binding with a type of **ncontext** when the type **nobject** was required.

13.2.2.2 CannotProceed

```
exception CannotProceed {
    NamingContext cxt;

    Name rest_of_name;
};
```

This exception is raised when an implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.

The **cxt** member contains the context that the operation may be able to retry from.

The **rest_of_name** member contains the remainder of the non-working name.

13.2.2.3 InvalidName

```
exception InvalidName {};
```

This exception is raised if a **Name** is invalid. A name of length zero is invalid (containing no name components). Implementations may place further limitations on what constitutes a legal name and raise this exception to indicate a violation.

13.2.2.4 AlreadyBound

exception AlreadyBound {};

Indicates an object is already bound to the specified name. Only one object can be bound to a particular **Name** in a context.

13.2.2.5 NotEmpty

exception NotEmpty {};

This exception is raised by **destroy** if the **NamingContext** contains bindings. A **NamingContext** must be empty to be destroyed.

13.2.3 Binding Objects

The binding operations name an object in a naming context. Once an object is bound, it can be found with the **resolve** operation. The Naming Service supports four operations to create bindings: **bind**, **rebind**, **bind_context**, and **rebind_context**. **bind_new_context** also creates a binding, see Section 13.2.6, “Creating Naming Contexts,” on page 223.

```
void bind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);
void bind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName);
```

13.2.3.1 bind

Creates an **nobject** binding in the naming context.

13.2.3.2 rebind

Creates an **nobject** binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type **nobject**; otherwise, a NotFound exception with a **why** reason of **not_object** is raised.

13.2.3.3 bind_context

Creates an **ncontext** binding in the parent naming context. Attempts to bind a nil context raise a BAD_PARAM exception.

13.2.3.4 rebind_context

Creates an **ncontext** binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type **ncontext**; otherwise, a NotFound exception with a **why** reason of **not_context** will be raised.

13.2.3.5 Usage

If a binding with the specified name already exists, **bind** and **bind_context** raise an AlreadyBound exception.

If an implementation places limits on the number of bindings within a context, **bind** and **bind_context** raise the IMP_LIMIT system exception if the new binding cannot be created.

Naming contexts bound using **bind_context** and **rebind_context** participate in name resolution when compound names are passed to be resolved; naming contexts bound with **bind** and **rebind** do not.

Use of **rebind_context** may leave a potential orphaned context (one that is unreachable within an instance of the Name Service). Policies and administration tools regarding potential orphan contexts are implementation-specific.

If **rebind** or **rebind_context** raise a NotFound exception because an already existing binding is of the wrong type, the **rest_of_name** member of the exception has a sequence length of 1.

13.2.4 Resolving Names

The **resolve** operation is the process of retrieving an object bound to a name in a given context. The given name must exactly match the bound name. The naming service does not return the type of the object. Clients are responsible for "narrowing" the object to the appropriate type. That is, clients typically cast the returned object from **Object** to a more specialized interface. The IDL definition of the **resolve** operation is:

```
Object resolve (in Name n)
                raises (NotFound, CannotProceed, InvalidName);
```

Names can have multiple components; therefore, name resolution can traverse multiple contexts. These contexts can be federated between different Naming Service instances.

13.2.5 Unbinding Names

The **unbind** operation removes a name binding from a context. The definition of the **unbind** operation is:

```
void unbind(in Name n)
        raises (NotFound, CannotProceed, InvalidName);
```

13.2.6 Creating Naming Contexts

The Naming Service supports two operations to create new contexts: **new_context** and **bind_new_context**.

```
NamingContext new_context();
NamingContext bind_new_context(in Name n)
        raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

13.2.6.1 new_context

This operation returns a new naming context. The new context is not bound to any name.

13.2.6.2 bind_new_context

This operation creates a new context and creates an **ncontext** binding for it using the name supplied as an argument.

13.2.6.3 Usage

If an implementation places limits on the number of naming contexts, both **new_context** and **bind_new_context** can raise the IMP_LIMIT system exception if the context cannot be created. **bind_new_context** can also raise IMP_LIMIT if the bind would cause an implementation limit on the number of bindings in a context to be exceeded.

13.2.7 Deleting Contexts

The **destroy** operation deletes a naming context.

```
void destroy()
    raises(NotEmpty);
```

This operation destroys its naming context. If there are bindings denoting the destroyed context, these bindings are *not* removed. If the naming context contains bindings, the operation raises NotEmpty.

13.2.8 Listing a Naming Context

The **list** operation allows a client to iterate through a set of bindings in a naming context.

```
void list (in unsigned long how_many,
        out BindingList bl, out BindingIterator bi);
};
```

list returns the bindings contained in a context in the parameter **bl**. The **bl** parameter is a sequence where each element is a **Binding** containing a **Name** of length 1 representing a single **NameComponent**.

The **how_many** parameter determines the maximum number of bindings to return in the parameter **bl**, with any remaining bindings to be accessed through the returned **BindingIterator bi**.

- A non-zero value of **how_many** guarantees that **bl** contains at most **how_many** elements. The implementation is free to return fewer than the number of bindings requested by **how_many**. However, for a non-zero value of **how_many**, it may not return a **bl** sequence with zero elements unless the context contains no bindings.
- If **how_many** is set to zero, the client is requesting to use only the **BindingIterator bi** to access the bindings and **list** returns a zero length sequence in **bl**.
- The parameter **bi** returns a reference to an iterator object.
- If the **bi** parameter returns a non-nil reference, this indicates that the call to **list** may not have returned all of the bindings in the context and that the remaining bindings (if any) must be retrieved using the iterator. This applies for all values of **how_many**.

- If the **bi** parameter returns a nil reference, this indicates that the **bl** parameter contains all of the bindings in the context. This applies for all values of **how_many**.

13.3 The BindingIterator Interface

The **BindingIterator** interface allows a client to iterate through the bindings using the **next_one** or **next_n** operations:

If a context is modified in between calls to **list**, **next_one**, or **next_n**, the behavior of further calls to **next_one** or **next_n** is implementation-dependent.

```
interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
                    out BindingList bl);
    void destroy();
};
```

13.3.1 Operations

13.3.1.1 next_one

The **next_one** operation returns the next binding. It returns true if it is returning a binding, false if there are no more bindings to retrieve. If **next_one** returns false, the returned **Binding** is indeterminate.

Further calls to **next_one** after it has returned false have undefined behavior.

13.3.1.2 next_n

next_n returns, in the parameter **bl**, bindings not yet retrieved with **list** or previous calls to **next_n** or **next_one**. It returns true if **bl** is a non-zero length sequence; it returns false if there are no more bindings and **bl** is a zero-length sequence.

The **how_many** parameter determines the maximum number of bindings to return in the parameter **bl**:

- A non-zero value of **how_many** guarantees that **bl** contains at most **how_many** elements. The implementation is free to return fewer than the number of bindings requested by **how_many**. However, it may not return a **bl** sequence with zero elements unless there are no bindings to retrieve.
- A zero value of **how_many** is illegal and raises a BAD_PARAM system exception.

next_n returns false with a **bl** parameter of length zero once all bindings have been retrieved. Further calls to **next_n** after it has returned a zero-length sequence have undefined behavior.

13.3.1.3 destroy

The **destroy** operation destroys its iterator. If a client invokes any operation on an iterator after calling **destroy**, the operation raises OBJECT_NOT_EXIST.

13.3.2 Garbage Collection of Iterators

Clients that create iterators but never call **destroy** can cause an implementation to permanently run out of resources. To protect itself against this scenario, an implementation is free to destroy an iterator object at any time without warning, using whatever algorithm it considers appropriate to choose iterators for destruction. In order to be robust in the presence of garbage collection, clients should be written to expect OBJECT_NOT_EXIST from calls to an iterator and handle this exception gracefully.

13.4 Stringified Names

Names are sequences of name components. This representation makes it difficult for applications to conveniently deal with names for I/O purposes, human or otherwise. This specification defines a syntax for stringified names and provides operations to convert a name in stringified form to its equivalent sequence form and vice-versa (see Section 13.5.4, “Converting between CosNames, Stringified Names, and URLs,” on page 230).

A stringified name represents one and only one **CosNaming::Name**. If two names are equal, their stringified representations are equal (and vice-versa).

The stringified name representation reserves use of the characters ‘/’, ‘.’, and ‘\’. The forward slash ‘/’ is a name component separator; the dot ‘.’ separates **id** and **kind** fields. The backslash ‘\’ is an escape character (see Section 13.4.2, “Escape Mechanism,” on page 227).

13.4.1 Basic Representation of Stringified Names

A stringified name consists of the name components of a name separated by a ‘/’ character. For example, a name consisting of the components “a,” “b,” and “c” (in that order) is represented as

a/b/c

Stringified names use the ‘.’ character to separate **id** and **kind** fields in the stringified representation. For example, the stringified name

a.b/c.d/.

represents the **CosNaming::Name**:

Index	**id**	**kind**
0	**a**	**b**
1	**c**	**d**
2	**<empty>**	**<empty>**

The single ‘.’ character is the only representation of a name component with empty **id** and **kind** fields.

If a name component in a stringified name does not contain a ‘.’ character, the entire component is interpreted as the **id** field, and the **kind** field is empty. For example:

a/./c.d/.e

corresponds to the **CosNaming::Name**:

Index	id	kind
0	**a**	**<empty>**
1	**<empty>**	**<empty>**
2	**c**	**d**
3	**<empty>**	**e**

If a name component has a non-empty **id** field and an empty **kind** field, the stringified representation consists only of the **id** field. A trailing '.' character is not permitted.

13.4.2 Escape Mechanism

The backslash '\' character escapes the reserved meaning of '/', '.', and '\' in a stringified name. The meaning of any other character following a '\' is reserved for future use.

13.4.2.1 NameComponent Separators

If a name component contains a '/' slash character, the stringified representation uses the '\' character as an escape. For example, the stringified name

a/x\/y\/z/b

represents the name consisting of the name components "a," "x/y/z," and "b."

13.4.2.2 Id and kind Fields

The backslash escape mechanism is also used for '.', so **id** and **kind** fields can contain a literal '.'. To illustrate, the stringified name

a\.b.c\.d/e.f

represents the **CosNaming::Name**:

Index	id	kind
0	**a.b**	**c.d**
1	**e**	**f**

13.4.2.3 The Escape Character

The escape character '\' must be escaped if it appears in a name component. For example, the stringified name:

a/b\\/c

represents the name consisting of the components "**a**," "**b**," and "**c**."

13.5 URL Schemes

This section describes the Uniform Resource Locator (URL) schemes available to represent a CORBA object and a CORBA object bound in a **NamingContext**.

13.5.1 IOR

The string form of an IOR (**IOR:<hex_octets>**) is a valid URL. The scheme name is **IOR** and the text after the ‘:’ is defined in the CORBA 2.3 specification, Section 13.6.6. The IOR URL is robust and insulates the client from the encapsulated transport information and object key used to reference the object. This URL format is independent of Naming Service.

13.5.2 corbaloc

It is difficult for humans to exchange IORs through non-electronic means because of their length and the text encoding of binary information. The **corbaloc** URL scheme provides URLs that are familiar to people and similar to **ftp** or **http** URLs.

The **corbaloc** URL is described in the CORBA 2.3 Specification, Section 13.6.6. This URL format is independent of the Naming Service.

13.5.3 corbaname

A **corbaname** URL is similar to a **corbaloc** URL. However a corbaname URL also contains a stringified name that identifies a binding in a naming context.

13.5.3.1 corbaname **Examples**

corbaname::555xyz.com/dev/NContext1#a/b/c

This example denotes a naming context that can be contacted in the same manner as a **corbaloc** URL at 555xyz.com with a key of “dev/NContext1”. The “#” character denotes the start of the stringified name, “**a/b/c**” . This name is resolved against the context to yield the final object.

corbaname::555xyz.com#a/b/c

When an object key is not specified, as in the above example, the default key of “NameService” is used to contact the naming context.

corbaname:rir:#a/b/c

This URL will resolve the stringified name “a/b/c” against the naming context returned by **resolve_initial_references(“NameService”)**.

corbaname:rir:

corbaname:rir:/NameService

The above URLs are equivalent to **corbaloc:rir:** and reference the naming context returned by **resolve_initial_references(“NameService”)**.

corbaname:atm:00033...#a/b/c

corbaname::55xyz.com,atm:00033.../dev/NCtext#a/b/c

These last URLs illustrate support of multiple protocols as allowed by **corbaloc** URLs. **atm:** is an example only and is not a defined URL protocol at this time.

Note – Unlike stringified names, **corbaname**s cannot be compared directly for equality as the address specification can differ for **corbaname** URLs with the same meaning.

13.5.3.2 corbaname Syntax

The full **corbaname** BNF is:

```
<corbaname>     = “corbaname:”<corbaloc_obj>[“#”<string_name>]
<corbaloc_obj>  = <obj_addr_list> [“/”<key_string>]
<obj_addr_list> = as defined in a corbaloc URL
<key_string>    = as defined in a corbaloc URL
<string_name>= stringified Name | empty_string
```

Where:

corbaloc_obj: portion of a corbaname URL that identifies the naming context. The syntax is identical to its use in a corbaloc URL.

obj_addr_list: as defined in a corbaloc URL.

key_string: as defined in a corbaloc URL.

string_name: a stringified Name with URL escapes as defined below.

13.5.3.3 corbaname Character Escapes

corbaname URLs use the escape mechanism described in the Internet Engineering Task Force (IETF) RFC 2396. These escape rules insure that URLs can be transferred via a variety of transports without undergoing changes. The character escape rules for the stringified name portion of a **corbaname** are:

US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

“;” | “/” | “:” | “?” | “@” | “&” | “=” | “+” | “$” |

“,” | “-” | “_” | ”.” | “!” | “~” | “*” | “’” | “(“ | “)”

13.5.3.4 corbaname Escape Mechanism

The percent ‘%’ character is used as an escape. If a character that requires escaping is present in a name component it is encoded as two hexadecimal digits following a “%” character to represent the octet. (The first hexadecimal character represent the high-order nibble of the octet, the second hexadecimal character represents the low-order nibble.) If a ‘%’ is not followed by two hex digits, the stringified name is syntactically invalid.

13.5.3.5 Examples

Stringified Name	After URL Escapes	Comment
a.b/c.d	a.b/c.d	URL form identical
<a>.b/c.d	%3ca%3e.b/c.d	Escaped “<“ and “>”
a.b/ c.d	a.b/%20%20c.d	Escaped two “ “ spaces
a%b/c%d	a%25b/c%25d	Escaped two “%” percents
a\\b/c.d	a%5c%5cb/c.d	Escaped “\” character, which is already escaped in the stringified name

13.5.3.6 corbaname Resolution

corbaname resolution can be implemented as a simple extension to **corbaloc** URL processing. Given a **corbaname**:

corbaname:<corbaloc_obj>[“#” <string_name>]

The **corbaname** is resolved by:

1. First constructing a **corbaloc** URL of the form:
   corbaloc:<corbaloc_obj>.

   If the **<corbaloc_obj>** does not contain a key string, a default key of “NameService” is used.

2. This is converted to a naming context object reference with **CORBA::ORB::string_to_object**.

3. The **<string_name>** is converted to a **CosNaming::Name**.

4. The resulting name is passed to a **resolve** operation on the naming context.

5. The object reference returned by the **resolve** is the result.

Implementations are not required to use the method described and may make optimizations appropriate to their environment.

13.5.4 Converting between CosNames, Stringified Names, and URLs

The **NamingContextExt** interface, derived from **NamingContext**, provides the operations required to use URLs and stringified names.

```
module CosNaming {
    // ...
    interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName          to_string(in Name n) raises(InvalidName);
        Name                to_name(in StringName sn)
                        raises(InvalidName);
```

```
            exception InvalidAddress {};

            URLString            to_url(in Address addr, in StringName sn)
                                raises(InvalidAddress, InvalidName);

            Object        resolve_str(in StringName sn)
                                raises(
                                    NotFound, CannotProceed,
                                    InvalidName
            );
        };
};
```

13.5.4.1 to_string

This operation accepts a **Name** and returns a stringified name. If the **Name** is invalid, an InvalidName exception is raised.

13.5.4.2 to_name

This operation accepts a stringified name and returns a **Name**. If the stringified name is syntactically malformed or violates an implementation limit, an InvalidName exception is raised.

13.5.4.3 resolve_str

This is a convenience operation that performs a resolve in the same manner as **NamingContext::resolve**. It accepts a stringified name as an argument instead of a **Name**.

13.5.4.4 to_url

This operation takes a corbaloc URL **<address>** and **<key_string>** component such as

- **:myhost.555xyz.com**
- **:myhost.555xyz.com/a/b/c**
- **atm:00002112...,:myhost.xyz.com/a/b/c**

for the first parameter, and a stringified name for the second. It then performs any escapes necessary on the parameters and returns a fully formed URL string. An exception is raised if either the corbaloc address and key parameter or name parameter are malformed.

It is legal for the stringified_name to be empty. If the address is empty, an InvalidAddress exception is raised.

13.5.4.5 URL to Object Reference

Conversions from URLs to objects are handled by **CORBA::ORB::string_to_object** as described in the CORBA 2.3 Specification, Section 13.6.6.

13.6 Initial Reference to a NamingContextExt

An initial reference to an instance of this interface can be obtained by calling **resolve_initial_references** with an **ObjectID** of **NameService**.

13.7 Consolidated IDL

```
// File: CosNaming.idl
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_

#pragma prefix "omg.org"

module CosNaming {
    typedef string Istring;

    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;

    enum BindingType { nobject, ncontext };

    struct Binding {
        Name            binding_name;
        BindingType     binding_type;
    };

    // Note: In struct Binding, binding_name is incorrectly defined
    // as a Name instead of a NameComponent. This definition is
    // unchanged for compatibility reasons.
    typedef sequence <Binding> BindingList;

    interface BindingIterator;

    interface NamingContext {
        enum NotFoundReason {
            missing_node, not_context, not_object
        };

        exception NotFound {
            NotFoundReason      why;
            Name                rest_of_name;
        };

        exception CannotProceed {
            NamingContext       cxt;
            Name                rest_of_name;
        };

        exception InvalidName{};

        exception AlreadyBound {};

        exception NotEmpty{};

        void          bind(in Name n, in Object obj)
            raises(
                NotFound, CannotProceed,
                InvalidName, AlreadyBound
            );

        void          rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);

        void          bind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed,
                 InvalidName, AlreadyBound
            );

        void          rebind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName);

        Object        resolve (in Name n)
            raises(NotFound, CannotProceed, InvalidName);

        void          unbind(in Name n)
            raises(NotFound, CannotProceed, InvalidName);

        NamingContext     new_context();
        NamingContext     bind_new_context(in Name n)
            raises(
                NotFound, AlreadyBound,
                CannotProceed, InvalidName
        );

        void          destroy() raises(NotEmpty);

        void          list(
                in unsigned long        how_many,
                out BindingList         bl,
                out BindingIterator     bi
        );

    };

    interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long, how_many, out BindingList bl);
        void          destroy();
    };

    interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName     to_string(in Name n) raises(InvalidName);
        Name           to_name(in StringName sn)
                           raises(InvalidName);

        exception InvalidAddress {};

        URLString      to_url(in Address addr, in StringName sn)
                           raises(InvalidAddress, InvalidName);

        Object         resolve_str(in StringName sn)
                           raises(
                               NotFound, CannotProceed,
                               InvalidName
                               );
    };
};
#endif // _COSNAMING_IDL_
```

14 Event Service

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. There are no requirements for conformant implementations of the CORBA/*e* Micro Profile in this chapter.

14.1 Overview

A standard CORBA request results in the synchronous execution of an operation by an object. If the operation defines parameters or return values, data is communicated between the client and the server. A request is directed to a particular object. For the request to be successful, both the client and the server must be available. If a request fails because the server is unavailable, the client receives an exception and must take some appropriate action.

In some scenarios, a more decoupled communication model between objects is required. For example:

- A system administration tool is interested in knowing if a disk runs out of space. The software managing a disk is unaware of the existence of the system administration tool. The software simply reports that the disk is full. When a disk runs out of space, the system administration tool opens a window to inform the user which disk has run out of space.
- A property list object is associated with an application object. The property list object is physically separate from the application object. The application object is interested in the changes made to its properties by a user. The properties can be changed without involving the application object. That is, in order to have reasonable response time for the user, changing a property does not activate the application object. However, when the application object is activated, it needs to know about the changes to its properties.
- A CASE tool is interested in being notified when a source program has been modified. The source program simply reports when it is modified. It is unaware of the existence of the CASE tool. In response to the notification, the CASE tool invokes a compiler.
- Several documents are linked to a spreadsheet. The documents are interested in knowing when the value of certain cells have changed. When the cell value changes, the documents update their presentations based on the spreadsheet. Furthermore, if a document is unavailable because of a failure, it is still interested in any changes to the cells and wants to be notified of those changes when it recovers.

14.1.1 Event Communication

The Event Service decouples the communication between objects. The Event Service defines two roles for objects: the supplier role and the consumer role. *Suppliers* produce event data and *consumers* process event data. Event data are communicated between suppliers and consumers by issuing standard CORBA requests.

There are two approaches to initiating event communication between suppliers and consumers, and two orthogonal approaches to the form that the communication can take.

The two approaches to initiating event communication are called the *push model* and the *pull model*. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

The communication itself can be either generic or typed. In the generic case, all communication is by means of generic `push` or `pull` operations that take a single parameter that packages all the event data. In the typed case, communication is via operations defined in OMG IDL. Event data is passed by means of the parameters, which can be defined in any manner desired.

An *event channel* is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests.

14.1.2 Example Scenario

This section provides a general scenario that illustrates how the Event Service can be used.

The Event Service can be used to provide "change notification." When an object is changed (its state is modified), an event can be generated that is propagated to all interested parties. For example, when a spreadsheet cell object is modified, all compound documents which contain a reference (link) to that cell can be notified (so the document can redisplay the referenced cell, or recalculate values that depend on the cell). Similarly, when an engineering specification object is modified, all engineers who have registered an interest in the specification can be notified that the specification has changed.

In this scenario, objects that can be "changed" act as suppliers, parties interested in receiving notifications of changes act as consumers, and one or more event channel objects are used as intermediaries between consumers and suppliers. *Either the push or the pull model can be used at either end.*

If the push model is used by suppliers, objects that can be changed support the **PushSupplier** interface so that event communication can be discontinued. Use the **EventChannel**, the **SupplierAdmin**, and the **ProxyPushConsumer** interfaces to register as suppliers of events, and use the **ProxyPushConsumer** interface to push events to event channels.

When a change occurs to an object, a changeable object invokes a `push` operation on the channel. It provides as an argument to the `push` operation information that describes the event. This information is of data type `any` - it can be as simple or as complex as is necessary. For example, the event information might identify the object reference of the object that has been changed, it might identify the kind of change that has occurred, it might provide a new displayable image of the changed object or it might identify one or more additional objects that describe the change that has been made.

If the pull model is used by consumers, all client objects that want to be notified of changes support the **PullConsumer** interface so communication can be discontinued, using the **EventChannel**, **ConsumerAdmin**, and **ProxyPullSupplier** interfaces to register as consumers of events, and using the **ProxyPullSupplier** interface to pull events from event channels.

The consumer may use either a blocking or non-blocking mechanism for receiving notification of changes. Using the **try_pull** operation, the consumer can periodically poll the channel for events. Alternatively, the consumer can use the `pull` operation which will block the consumer's execution thread until an event is generated by some supplier.

Event channels act as the intermediaries between the objects being changed and objects interested in knowing about changes. The channels that provide change notification can be general purpose, well-known objects (e.g., "persistent server-based objects" that are run as part of a workgroup-wide framework of objects that provide "desktop services") or specific-to-task objects (e.g., temporary objects that are created when needed). Objects that use event channels may locate the channels by looking for them in a persistently available server (e.g., by looking for them in a naming service) or they may be given references to these objects as part of a specific-to-task object protocol (e.g., when an "open" operation is invoked on an object, the object may return the reference to an event channel which the caller should use until the object is closed).

Event channels determine how changes are propagated between suppliers and consumers (i.e., the qualities of service). For example, an event channel determines the persistence of an event. The channel may keep an event for a specified period of time, passing it along to any consumer who registers with the channel during that period of time (e.g., it may keep event notifications about changes to engineering specifications for a week). Alternatively, the channel may only pass on events to consumers who are currently waiting for notification of changes (e.g., notifications of changes to a spreadsheet cell may only be sent to consumers who are currently displaying that cell).

This scenario exemplifies one way the event service described here forms a basic building block used in providing higher-level services specific to an application or common facilities framework of objects.

Instead of using the generic event channel, a typed event channel could also have been used.

14.1.3 Design Principles

The Event Service design satisfies the following principles:

- Events work in a distributed environment. The design does not depend on any global, critical, or centralized service.
- Event services allow multiple consumers of an event and multiple event suppliers.
- Consumers can either request events or be notified of events, whichever is more appropriate for application design and performance.
- Consumers and suppliers of events support standard OMG IDL interfaces; no extensions to CORBA are necessary to define these interfaces.
- A supplier can issue a single standard request to communicate event data to all consumers at once.
- Suppliers can generate events without knowing the identities of the consumers. Conversely, consumers can receive events without knowing the identities of the suppliers.
- The Event Service interfaces allow multiple qualities of service, for example, for different levels of reliability. It also allows for future interface extensions, such as for additional functionality.
- The Event Service interfaces are capable of being implemented and used in different operating environments, for example, in environments that support threading and those that do not.

14.1.4 Resolution of Technical Issues

This specification addresses the issues identified for event services in the OMG *Object Services Architecture*[1] document as follows:

- **Distributed environment:** The interfaces are designed to allow consumers and suppliers of events to be disconnected from time to time, and do not require centralized event identification, processing, routing, or other services that might be a bottleneck or a single point of failure.

  Events themselves are *not* objects because the CORBA distributed object model does not support passing objects by value.
- **Event generation:** The specification describes how events are generated and delivered in a very general fashion, with

[1] *Object Services Architectur*e, Document Number 92-8-4, Object Management Group, Framingham, MA, 1992.

event channels as intermediate routing points. It does not require (or preclude) polling, nor does it require that an event supplier directly notify every interested party.

- **Events involving multiple objects:** Complex events may be handled by constructing a notification tree of event consumer/suppliers checking for successively more specific event predicates. The specification does not require a general or global event predicate evaluation service as this may not be sufficiently reliable, efficient, or secure in a distributed, heterogeneous (potentially decoupled) environment.
- **Scoping, grouping, and filtering events:** The specification takes advantage of CORBA's distributed scoping and grouping mechanisms for the identifier and type of events. Event filtering is easily achieved through event channels that selectively deliver events from suppliers to consumers. Event channels can be composed; that is, one event channel can consume events supplied by another. Typed event channels can provide filtering based on event type.
- **Registration and generation of events:** Consumers and suppliers register with event channels themselves. Event channels are objects and they are found by any fashion that objects can be found. A global registration service is not required; any object that conforms to the IDL interface may consume an event.
- **Event parameters:** The specification supports a parameter of type `any` that can be delivered with an event, used for application-specific data.
- **Forgery and secure events:** Because event suppliers are objects, the specification leverages any ORB work on security for object references and communication.
- **Performance:** The design is a minimalist one, and requires only one ORB call per event received. It supports both push-style and pull-style notification to avoid inefficient event polling. Since event suppliers, consumers, and channels are all ORB objects, the service directly benefits from a Library Object Adapter or any other ORB optimizations.
- **Formalized Event Information:** For specific application environments and frameworks it may be beneficial to formalize the data associated with an event (defined in this specification as type any). This can be accomplished by defining a typed structure for this information. Depending on the needs of the environment, the kinds of information included might be a priority, timestamp, origin string, and confirmation indicator. This information might be solely for the benefit of the event consumer or might also be interpreted by particular event channel implementations.
- **Confirmation of Reception:** Some applications may require that consumers of an event provide an explicit confirmation of reception back to the supplier. This can be supported effectively using a "reverse" event channel through which consumers send back confirmations as normal events. This obviates the need for any special confirmation mechanism. However, strict atomic delivery between all suppliers and all consumers requires additional interfaces.

14.1.5 Quality of Service

Application domains requiring event-style communication have diverse reliability requirements, from "at-most-once" semantics (best effort) to guaranteed "exactly-once" semantics, availability requirements, throughput requirements, performance requirements (i.e., how fast events are disseminated), and scalability requirements.

Clearly no single implementation of the Event Service can optimize such a diverse range of technical requirements. Hence, multiple implementations of event services are to be expected, with different services targeted toward different environments. As such, the event interfaces do not dictate *qualities of service*. Different implementations of the Event Service interfaces can support different qualities of service to meet different application needs.

For example, an implementation that trades at most once delivery to a single consumer in favor of performance is useful for some applications; an implementation that favors performance but cannot preclude duplicate delivery is useful for other applications. Both are acceptable implementations of the interfaces described in this chapter.

Clearly, an implementation of an event channel that discards all events is *not a useful* implementation. Useful implementations will at least support “best-effort” delivery of events.

Note that the interfaces defined in this chapter are incomplete for implementations that support strict notions of atomicity. That is, additional interfaces are needed by an implementation to guarantee that either all consumers receive an event or none of the consumers receive an event; and that all events are received in the same order by all consumers.

14.1.6 Generic Event Communication

There are two basic models for communicating event data between suppliers and consumers: the *push model* and the *pull model*.

14.1.6.1 Push Model

In the push model, suppliers “push” event data to consumers; that is, suppliers communicate event data by invoking `push` operations on the **PushConsumer** interface.

To set up a push-style communication, consumers and suppliers exchange **PushConsumer** and **PushSupplier** object references. Event communication can be broken by invoking a **disconnect_push_consumer** operation on the **PushConsumer** interface or by invoking a **disconnect_push_supplier** operation on the **PushSupplier** interface. If the **PushSupplier** object reference is nil, the connection cannot be broken via the supplier.

Figure 14.1 illustrates push-style communication between a supplier and a consumer.

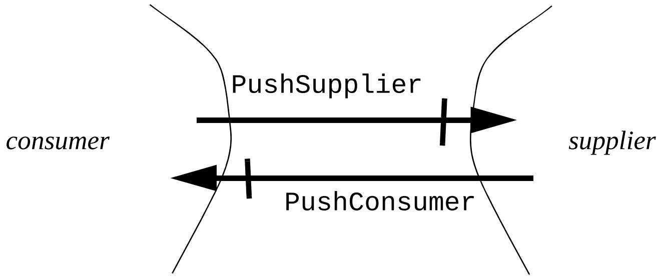

Figure 14.1- Push-style Communication Between a Supplier and a Consumer

14.1.6.2 Pull Model

In the pull model, consumers “pull” event data from suppliers; that is, consumers request event data by invoking **pull** operations on the **PullSupplier** interface.

To set up a pull-style communication, consumers and suppliers must exchange **PullConsumer** and **PullSupplier** object references. Event communication can be broken by invoking a **disconnect_pull_consumer** operation on the **PullConsumer** interface or by invoking a **disconnect_pull_supplier** operation on the **PullSupplier** interface. If the **PullConsumer** object reference is nil, the connection cannot be broken via the consumer.

Figure 14.2 illustrates pull-style communication between a supplier and a consumer.

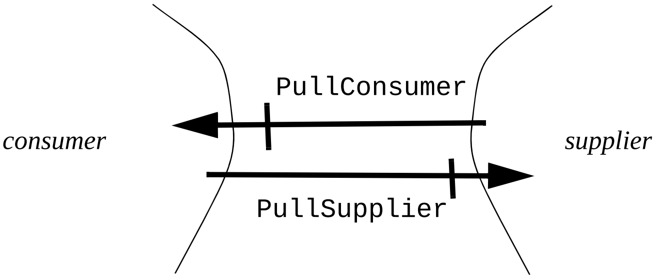

Figure 14.2- Pull-style Communication Between a Supplier and a Consumer

14.2 The CosEventComm Module

The communication styles shown in Chapter 1 are both supported by four simple interfaces: **PushConsumer**, **PushSupplier**, and **PullSupplier**, **PullConsumer**. These interfaces are defined in an OMG IDL module named **CosEventComm**, as shown below.

```
module CosEventComm {

    exception Disconnected{};

    interface PushConsumer {
        void push (in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };

   interface PushSupplier {
        void disconnect_push_supplier();
    };

    interface PullSupplier {
        any pull () raises(Disconnected);
        any try_pull (out boolean has_event)
            raises(Disconnected);
        void disconnect_pull_supplier();
    };

    interface PullConsumer {
        void disconnect_pull_consumer();
    };

};
```

14.2.1 The PushConsumer Interface

A push-style consumer supports the **PushConsumer** interface to receive event data.

```
interface PushConsumer {
    void push (in any data) raises(Disconnected);
    void disconnect_push_consumer();
};
```

A supplier communicates event data to the consumer by invoking the **push** operation and passing the event data as a parameter.

The **disconnect_push_consumer** operation terminates the event communication; it releases resources used at the consumer to support the event communication. The *PushConsumer* object reference is disposed. Calling **disconnect_push_consumer** causes the implementation to call the **disconnect_push_supplier** operation on the corresponding **PushSupplier** interface (if that interface is known).

14.2.2 The PushSupplier Interface

A push-style supplier supports the **PushSupplier** interface.

```
interface PushSupplier {
    void disconnect_push_supplier();
};
```

The **disconnect_push_supplier** operation terminates the event communication; it releases resources used at the supplier to support the event communication. The *PushSupplier* object reference is disposed. Calling **disconnect_push_supplier** causes the implementation to call the **disconnect_push_consumer** operation on the corresponding **PushConsumer** interface (if that interface is known).

14.2.3 The PullSupplier Interface

A pull-style supplier supports the **PullSupplier** interface to transmit event data.

```
interface PullSupplier {
    any pull () raises(Disconnected);
    any try_pull (out boolean has_event)
        raises(Disconnected);
    void disconnect_pull_supplier();
};
```

A consumer requests event data from the supplier by invoking either the **pull** operation or the **try_pull** operation on the supplier.

- The **pull** operation blocks until the event data is available or an exception is raised.[2] It returns the event data to the consumer.
- The **try_pull** operation does not block: if the event data is available, it returns the event data and sets the **has_event** parameter to *true*; if the event is not available, it sets the **has_event** parameter to *false* and the event data is returned as long with an undefined value.

[2] This, of course, may be a standard CORBA exception.

The **disconnect_pull_supplier** operation terminates the event communication; it releases resources used at the supplier to support the event communication. The **PullSupplier** object reference is disposed. Calling **disconnect_pull_supplier** causes the implementation to call the **disconnect_pull_consumer** operation on the corresponding **PullConsumer** interface (if that interface is known).

14.2.4 The PullConsumer Interface

A pull-style consumer supports the **PullConsumer** interface.

```
interface PullConsumer {
    void disconnect_pull_consumer();
};
```

The **disconnect_pull_consumer** operation terminates the event communication; it releases resources used at the consumer to support the event communication.

The **PullConsumer** object reference is disposed. Calling **disconnect_pull_consumer** causes the implementation to call the **disconnect_pull_supplier** operation on the corresponding **PullSupplier** interface (if that interface is known).

14.2.5 Disconnection Behavior

Calling a disconnect operation on a consumer or supplier interface may cause a call to the corresponding disconnect operation on the connected supplier or consumer. Implementations must take care to avoid infinite recursive calls to these disconnect operations. If a consumer or supplier has received a disconnect call and subsequently receives another disconnect call, it shall raise a CORBA::OBJECT_NOT_EXIST exception.

14.3 Event Channels

The *event channel* is a service that decouples the communication between suppliers and consumers. The event channel is itself both a consumer and a supplier of the event data.

An event channel can provide asynchronous communication of event data between suppliers and consumers. Although consumers and suppliers communicate with the event channel using standard CORBA requests, the event channel does not need to supply the event data to its consumer at the same time it consumes the data from its supplier.

14.3.1 Push-Style Communication with an Event Channel

The supplier pushes event data to the event channel; the event channel, in turn, pushes event data to the consumer. Figure 14.3 illustrates a push-style communication between a supplier and the event channel, and a consumer and the event channel.

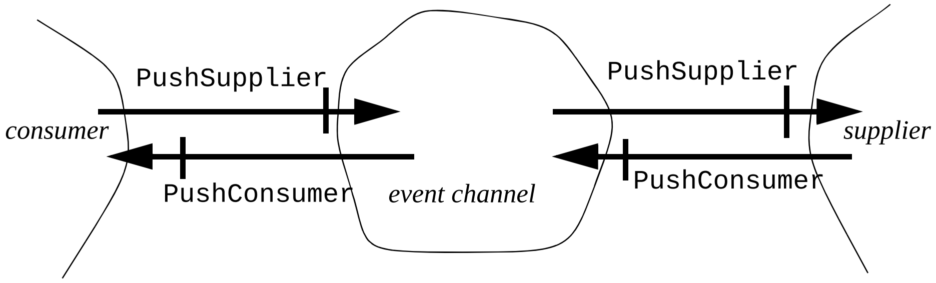

Figure 14.3- Push-style Communication Between a Supplier and an Event Channel, and a Consumer and an Event Channel

14.3.2 Pull-Style Communication with an Event Channel

The consumer pulls event data from the event channel; the event channel, in turn, pulls event data from the supplier. Figure 14.4 illustrates a pull-style communication between a supplier and the event channel, and a consumer and the event channel.

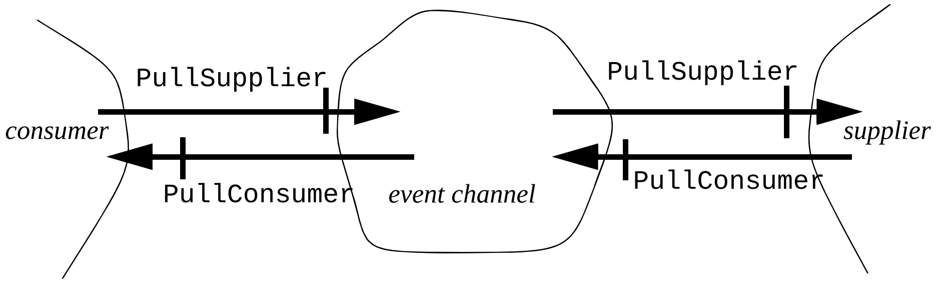

Figure 14.4- Pull-style communication between a supplier and an event channel and a consumer and the event channel

14.3.3 Mixed Style Communication with an Event Channel

An event channel can communicate with a supplier using one style of communication, and communicate with a consumer using a different style of communication.

Figure 14.5 illustrates a push-style communication between a supplier and an event channel, and a pull-style communication between a consumer and the event channel. The consumer pulls the event data that the supplier has pushed to the event channel.

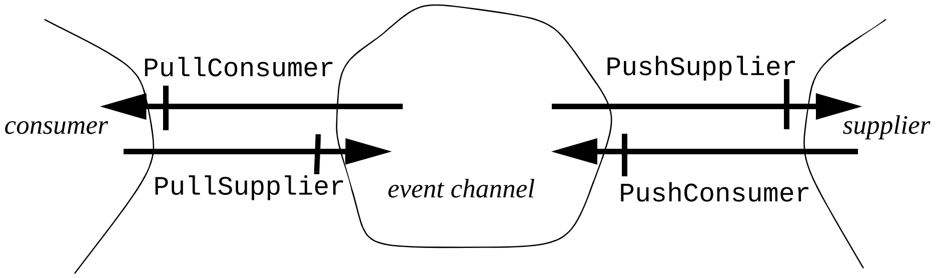

Figure 14.5- Push-style Communication Between a Supplier and an Event Channel, and Pull-style Communication Between a Consumer and an Event Channel

14.3.4 Multiple Consumers and Multiple Suppliers

Figure 14.3, Figure 14.4, and Figure 14.5 illustrate event channels with a single supplier and a single consumer. An event channel can also provide many-to-many communication. The channel consumes events from one or more suppliers, and supplies events to one or more consumers. Subject to the quality of service of a particular implementation, an event channel provides an event to all consumers.

Figure 14.6 illustrates an event channel with multiple push-style consumers and multiple push-style suppliers.

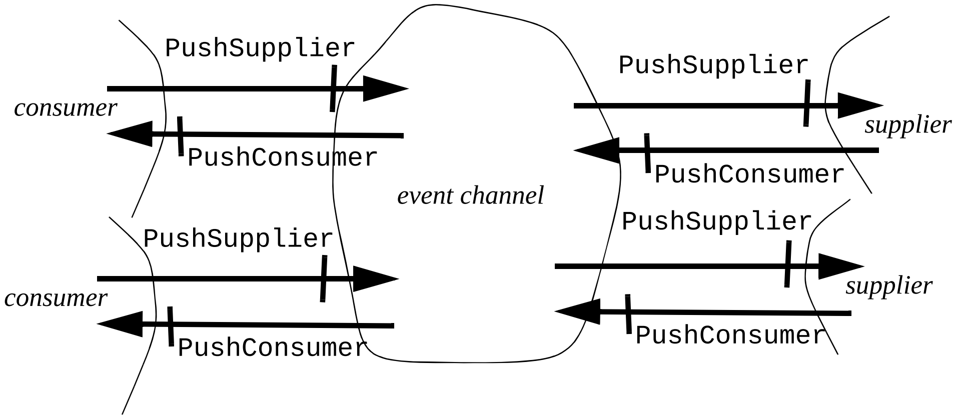

Figure 14.6- An Event Channel with Multiple Suppliers and Multiple Consumers

An event channel can support consumers and suppliers using different communication models.

If an event channel has pull suppliers, it continues to pull events from the suppliers, regardless of whether any consumers are connected to the channel.

14.3.5 Event Channel Administration

The event channel is built up incrementally. When an event channel is created, no suppliers or consumers are connected to the event channel. Upon creation of the channel, the factory returns an object reference that supports the **EventChannel** interface, as illustrated in Figure 14.7.

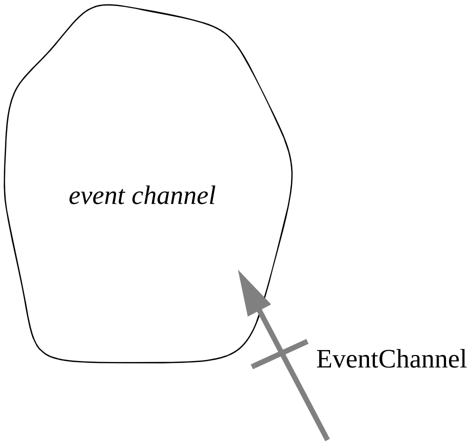

Figure 14.7- A newly created event channel. The channel has no suppliers or consumers.

The **EventChannel** interface defines three administrative operations: an operation returning a **ConsumerAdmin** object for adding consumers, an operation returning a **SupplierAdmin** object for adding suppliers, and an operation for destroying the channel.

The operations for adding consumers return *proxy suppliers*. A proxy supplier is similar to a normal supplier (in fact, it inherits the interface of a supplier), but includes an additional method for connecting a consumer to the proxy supplier.

The operations for adding suppliers return *proxy consumers*. A proxy consumer is similar to a normal consumer (in fact, it inherits the interface of a consumer), but includes an additional method for connecting a supplier to the proxy consumer.

Registration of a producer or consumer is a two step process. An event-generating application first obtains a proxy consumer from a channel, then “connects” to the proxy consumer by providing it with a supplier. Similarly, an event-receiving application first obtains a proxy supplier from a channel, then “connects” to the proxy supplier by providing it with a consumer.

The reason for the two-step registration process is to support composing event channels by an external agent. Such an agent would compose two channels by obtaining a proxy supplier from one and a proxy consumer from the other, and passing each of them a reference to the other as part of their connect operation.

Proxies are in one of three states: *disconnected*, *connected*, or *destroyed*. Figure 14.8 gives a state diagram for a proxy. The nodes of the diagram are the states and the edges are labeled with the operations that change the state of the proxy. **Push/pull** operations are only valid in the *connected* state.

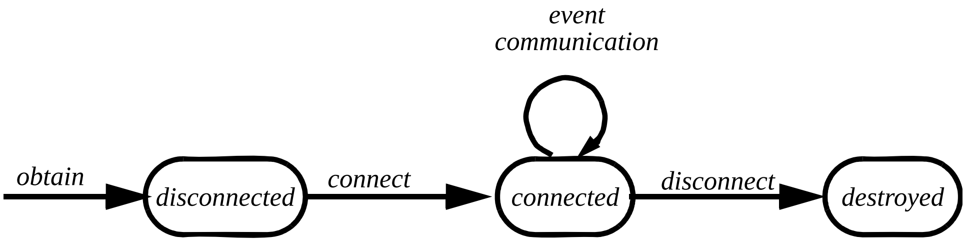

Figure 14.8- State diagram of a proxy

14.4 The CosEventChannelAdmin Module

The **CosEventChannelAdmin** module defines the interfaces for making connections between suppliers and consumers. The **CosEventChannelAdmin** module is defined below.

```
#include "CosEventComm.idl"

module CosEventChannelAdmin {

    exception AlreadyConnected {};
    exception TypeError {};

    interface ProxyPushConsumer: CosEventComm::PushConsumer {
        void connect_push_supplier(
                in CosEventComm::PushSupplier push_supplier)
            raises(AlreadyConnected);
    };

    interface ProxyPullSupplier: CosEventComm::PullSupplier {
        void connect_pull_consumer(
                in CosEventComm::PullConsumer pull_consumer)
            raises(AlreadyConnected);
    };

    interface ProxyPullConsumer: CosEventComm::PullConsumer {
        void connect_pull_supplier(
                in CosEventComm::PullSupplier pull_supplier)
            raises(AlreadyConnected,TypeError);
    };

    interface ProxyPushSupplier: CosEventComm::PushSupplier {
        void connect_push_consumer(
                in CosEventComm::PushConsumer push_consumer)
            raises(AlreadyConnected, TypeError);
    };
    interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier();
        ProxyPullSupplier obtain_pull_supplier();
    };

    interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer();
        ProxyPullConsumer obtain_pull_consumer();
    };

    interface EventChannel {
        ConsumerAdmin for_consumers();
        SupplierAdmin for_suppliers();
        void destroy();
    };

};
```

14.4.1 The EventChannel Interface

The **EventChannel** interface defines three administrative operations: adding consumers, adding suppliers, and destroying the channel.

```
interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
};
```

Any object that possesses an object reference that supports the **EventChannel** interface can perform these operations:

- The **ConsumerAdmin** interface allows consumers to be connected to the event channel. The **for_consumers** operation returns an object reference that supports the **ConsumerAdmin** interface.
- The **SupplierAdmin** interface allows suppliers to be connected to the event channel. The **for_suppliers** operation returns an object reference that supports the **SupplierAdmin** interface.
- The **destroy** operation destroys the event channel. Destroying an event channel destroys all **ConsumerAdmin** and **SupplierAdmin** objects that were created via that channel. Destruction of a **ConsumerAdmin** or **SupplierAdmin** object causes the implementation to invoke the disconnect operation on all proxies that were created via that **ConsumerAdmin** or **SupplierAdmin** object.

Consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers. For example, a creator might wish to be the sole supplier of event data but allow many consumers to be connected to the channel. In such a case, the creator would simply export the **ConsumerAdmin** object.

14.4.2 The ConsumerAdmin Interface

The **ConsumerAdmin** interface defines the first step for connecting consumers to the event channel; clients use it to obtain proxy suppliers.

```
interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ProxyPullSupplier obtain_pull_supplier();
};
```

The **obtain_push_supplier** operation returns a **ProxyPushSupplier** object. The **ProxyPushSupplier** object is then used to connect a push-style consumer.

The **obtain_pull_supplier** operation returns a **ProxyPullSupplier** object. The **ProxyPullSupplier** object is then used to connect a pull-style consumer.

14.4.3 The SupplierAdmin Interface

The **SupplierAdmin** interface defines the first step for connecting suppliers to the event channel; clients use it to obtain proxy consumers.

```
interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
};
```

The **obtain_push_consumer** operation returns a **ProxyPushConsumer** object. The **ProxyPushConsumer** object is then used to connect a push-style supplier.

The **obtain_pull_consumer** operation returns a **ProxyPullConsumer** object. The **ProxyPullConsumer** object is then used to connect a pull-style supplier.

14.4.4 The ProxyPushConsumer Interface

The **ProxyPushConsumer** interface defines the second step for connecting push suppliers to the event channel.

```
interface ProxyPushConsumer: CosEventComm::PushConsumer {
    void connect_push_supplier(
            in CosEventComm::PushSupplier push_supplier)
        raises(AlreadyConnected);
};
```

A nil object reference may be passed to the **connect_push_supplier** operation; if so a channel cannot invoke the **disconnect_push_supplier** operation on the supplier; the supplier may be disconnected from the channel without being informed. If a non-nil reference is passed to **connect_push_supplier**, the implementation calls **disconnect_push_supplier** via that reference when the **ProxyPushConsumer** is destroyed.

If the **ProxyPushConsumer** is already connected to a **PushSupplier**, then the AlreadyConnected exception is raised.

14.4.5 The ProxyPullSupplier Interface

The **ProxyPullSupplier** interface defines the second step for connecting pull consumers to the event channel.

```
interface ProxyPullSupplier: CosEventComm::PullSupplier {
    void connect_pull_consumer(
            in CosEventComm::PullConsumer pull_consumer)
        raises(AlreadyConnected);
};
```

A nil object reference may be passed to the **connect_pull_consumer** operation; if so a channel cannot invoke a **disconnect_pull_consumer** operation on the consumer; the consumer may be disconnected from the channel without being informed. If a non-nil reference is passed to **connect_pull_consumer**, the implementation calls **disconnect_pull_consumer** via that reference when the **ProxyPullSupplier** is destroyed.

If the **ProxyPullSupplier** is already connected to a **PullConsumer**, then the AlreadyConnected exception is raised.

14.4.6 The ProxyPullConsumer Interface

The **ProxyPullConsumer** interface defines the second step for connecting pull suppliers to the event channel.

```
interface ProxyPullConsumer: CosEventComm::PullConsumer {
    void connect_pull_supplier(
            in CosEventComm::PullSupplier pull_supplier)
        raises(AlreadyConnected, TypeError);
};
```

The implementation calls **disconnect_pull_supplier** on the reference passed to **connect_pull_supplier** when the **ProxyPullConsumer** is destroyed.

Implementations shall raise the CORBA standard BAD_PARAM exception if a nil object reference is passed to the **connect_pull_supplier** operation.

If the **ProxyPullConsumer** is already connected to a **PullSupplier**, then the AlreadyConnected exception is raised.

An implementation of a **ProxyPullConsumer** may put additional requirements on the interface supported by the pull supplier. If the pull supplier does not meet those requirements, the **ProxyPullConsumer** raises the TypeError exception.

14.4.7 The ProxyPushSupplier Interface

The **ProxyPushSupplier** interface defines the second step for connecting push consumers to the event channel.

```
interface ProxyPushSupplier: CosEventComm::PushSupplier {
    void connect_push_consumer(
            in CosEventComm::PushConsumer push_consumer)
        raises(AlreadyConnected, TypeError);
};
```

The implementation calls **disconnect_push_consumer** on the reference passed to **connect_push_consumer** when the **ProxyPushSupplier** is destroyed.

Implementations shall raise the CORBA standard BAD_PARAM exception if a nil object reference is passed to the **connect_push_consumer** operation.

If the **ProxyPushSupplier** is already connected to a **PushConsumer**, then the AlreadyConnected exception is raised.

An implementation of a **ProxyPushSupplier** may put additional requirements on the interface supported by the push consumer. If the push consumer does not meet those requirements, the **ProxyPushSupplier** raises the TypeError exception.

14.5 Consolidated IDL

14.5.1 CosEventComm

```
//File: CosEventComm.idl
//Part of the Event Service

#ifndef _COS_EVENT_COMM_IDL_
#define _COS_EVENT_COMM_IDL_
#pragma prefix "omg.org"

#if defined(CORBA_E_COMPACT)

module CosEventComm
{
# ifndef _PRE_3_0_COMPILER_
        typeprefix "omg.org";
# endif // _PRE_3_0_COMPILER_

    exception Disconnected{};

    interface PushConsumer
    {
        void push (in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };

  interface PushSupplier
    {
        void disconnect_push_supplier();
    };
};
#endif /* defined(CORBA_E_COMPACT) */
#endif /* ifndef _COS_EVENT_COMM_IDL_ */
```

14.5.2 CosEventChannelAdmin

```
// File: CosEventChannelAdmin.idl
// IDL
#ifndef _COS_EVENT_CHANNEL_ADMIN_IDL_
```

```
#define _COS_EVENT_CHANNEL_ADMIN_IDL_

#if defined(CORBA_E_COMPACT)

#include <CosEventComm.idl>
#pragma prefix "omg.org"
module CosEventChannelAdmin {
# ifndef _PRE_3_0_COMPILER_
   typeprefix "omg.org";
# endif // _PRE_3_0_COMPILER_

      exception AlreadyConnected {};
      exception TypeError {};

      interface ProxyPushConsumer: CosEventComm::PushConsumer {
            void connect_push_supplier(
                        in CosEventComm::PushSupplier push_supplier)
                  raises(AlreadyConnected);
      };

      interface ProxyPushSupplier: CosEventComm::PushSupplier {
            void connect_push_consumer(
                        in CosEventComm::PushConsumer push_consumer)
                  raises(AlreadyConnected, TypeError);
      };

      interface ConsumerAdmin {
            ProxyPushSupplier obtain_push_supplier();
      };

      interface SupplierAdmin {
            ProxyPushConsumer obtain_push_consumer();
      };

      interface EventChannel {
            ConsumerAdmin for_consumers();
            SupplierAdmin for_suppliers();
            void destroy();
      };
};
#endif /* if defined(CORBA_E_COMPACT) */
#endif /* ifndef _COS_EVENT_CHANNEL_ADMIN_IDL_ */
```

15 Lightweight Log Service

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. There are no requirements for conformant implementations of the CORBA/*e* Micro Profile in this chapter.

15.1 Overview

This specification defines a Lightweight Logging Service intended for use in resource-constraint systems like embedded and/or real-time CORBA systems.

The Lightweight Logging Service specification contained in this document is primarily intended as an efficient, central facility inside an embedded or real-time environment to accept and manage logging records. These records are emitted from applications residing in the same environment and stored in a memory-only storage area owned and managed by the Lightweight Logging Service. The service was designed to be a mostly compatible subset of the Telecom Log Service, however, it differs in the way logging records are written to the log; or looked up and retrieved from the log.

This service has wide application. It will find its way into all areas of embedded systems, like machine control, onboard vehicle systems, etc., but also into ubiquitous computing devices like pocket computer and electronic organizers. But also "regular" application areas will benefit, if just a small, memory-only logging facility is required.

15.2 Architecture

In consideration of the resource constraints imposed by the embedded system environment, the Lightweight Logging Service is a free-standing, self-contained service, and not connected to an event channel or similar infrastructure. The core of the Lightweight Logging Service is represented by the class **Log**, which encapsulates the storage area for logging records and provides the methods comprising the logging functionality. However, the class **Log** does not communicate directly with the rest of the environment. Communication with the surrounding environment is handled through three distinct interfaces.

LogProducer	This interface allows the insertion of new log records into the logging storage area encapsulated by the Log class. In favor of preserving the overall operational integrity of the system, no guarantee is made that a logging record is accepted and stored if the logging service is unable to process and /or store it.
LogConsumer	This interface allows the retrieval of logging records from the storage area encapsulated by the Log class.
LogAdmin	This interface provides the management functionality to operate and manage the logging service.

The above three interfaces are derived from an abstract super interface **LogStatus**, which provides informational functionality common to all three interfaces.

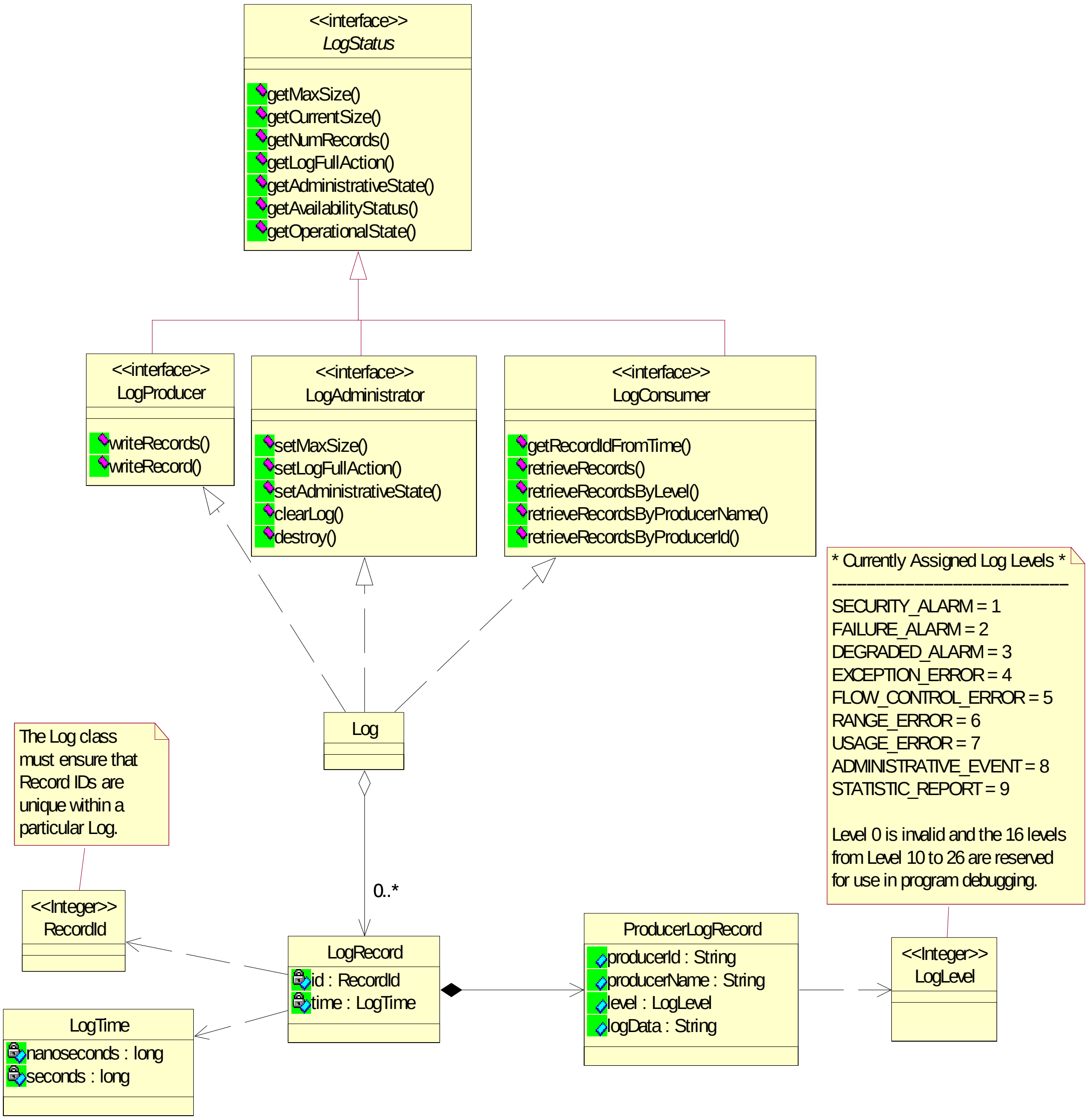

Figure 15.1- Lightweight Logging Service PIM

As shown in Figure 15.1, the central piece of the Lightweight Logging Service is the class Log, which encapsulates the storage area for logging records and provides all necessary operations to manage and operate the Lightweight Logging Service. Note, however, that the operations should not be directly accessible to any clients of the logging service. Instead, a set of interfaces is provided to give controlled access to each kind of clients. This is kind of a “poor man’s” protection system, which provides sufficient protection against accidental misuse, while, at the same time, giving tribute to the severe resource constraints common in embedded devices.

15.3 Types and Data Structures

15.3.1 InvalidParam Exception

```
exception InvalidParam { string details; };
```

The **InvalidParam** exception indicates that a provided parameter was invalid. Details about the cause for this exception are delivered in the string attribute **details**.

15.3.2 LogLevel

Type **LogLevel** is an enumeration-like type that is utilized to identify log levels.

```
unsigned short LogLevel;

const unsigned short SECURITY_ALARM = 1;
const unsigned short FAILURE_ALARM = 2;
const unsigned short DEGRADED_ALARM =3;
const unsigned short EXCEPTION_ERROR =4;
const unsigned short FLOW_CONTROL_ERROR =5;
const unsigned short RANGE_ERROR =6;
const unsigned short USAGE_ERROR = 7;
const unsigned short ADMINISTRATIVE_EVENT = 8;
const unsigned short STATISTIC_REPORT = 9;
// Values ranging from 10 to 26 are reserved for
// 16 debugging levels.
```

The **LogLevel** allows a classification of the logging record. The value provided is recorded in the logging record and provided to the consumer at retrieval, but it has no particular meaning or side effects during storage of the record in the Log.

15.3.3 OperationalState

```
enum OperationalState {disabled, enabled};
```

The enumeration `OperationalStateType` defines the Log states of operation. When the Log is **enabled** it is fully functional and is available for use by log producer and log consumer clients. A Log that is **disabled** has encountered a runtime problem and is not available for use by log producers or log consumers. The internal error conditions that cause the Log to set the operational state to **enabled** or **disabled** are implementation specific.

15.3.4 AdministrativeState

```
enum AdministrativeState {locked, unlocked};
```

The **AdministrativeState** type denotes the active logging state of an operational Log. When set to **unlocked** the Log will accept records for storage, per its operational parameters. When set to **locked** the Log will not accept new log records and records can be read or deleted only.

15.3.5 LogFullAction

```
enum LogFullAction {WRAP, HALT};
```

This type specifies the action that the Log should take when its internal buffers become full of data, leaving no room for new records to be written. **WRAP** indicates that the Log will overwrite the oldest **LogRecords** with the newest records, as they are written to the Log. The Log will overwrite as many of the oldest LogRecords as needed to accommodate the newest records. **HALT** indicates that the Log will stop logging when full.

15.3.6 LogAvailabilityStatus

```
struct AvailabilityStatus{
    boolean off_duty;
    boolean log_full;
};
```

The **AvailabilityStatus** denotes whether or not the Log is available for use. When **true**, `off_duty` indicates the Log is **locked** (administrative state) or **disabled** (operational state). When **true**, `log_full` indicates the Log storage is full.

Struct member	Description
off_duty	Indicates that the log is unavailable, if true.
log_full	Indicates that the log storage area is full, if true.

15.3.7 LogTime

```
struct LogTime {
    long seconds;
    long nanoseconds;
};
```

This type provides the time format used by the Log to time stamp LogRecords. Each field is intended to directly map to the POSIX timespec structure.

Note – An implementation should exclusively use UTC for time recording to support location transparency.

15.3.8 ProducerLogRecord

```
   struct ProducerLogRecord {
    string    producerId;
    string    producerName;
    LogLevel  level;
    string    logData;
};
typedef sequence <ProducerLogRecord>
ProducerLogRecordSequence;
```

Log producers format log records as defined in the structure **ProducerLogRecord**.

Struct member	Description
producerId	This field uniquely identifies the source of a log record. The value is the component's identifier and should be unique for each log record producing component within the Domain.
producerName	This field identifies the producer of a log record in textual format. This field is assigned by the log producer, thus is not unique within the Domain (e.g., multiple instances of an application will assign the same name to the ProducerName field).
level	The level field can be used to classify the log record according to the LogLevel type.
logData	This field contains the informational message being logged.

This structure represents a logging record written by a log producer client to the Log via the **LogProducer** interface. Upon reception, it is encapsulated by the **LogRecord** described in Section 15.3.2, "LogLevel," on page 255.

15.3.9 RecordId

```
typedef unsigned long long RecordId;
```

This type provides the unique record ID that is assigned to a **LogRecord** by the **Log**.

15.3.9.1 Mapping from the Platform Independent Model

This IDL type is the result of a one-to-one mapping from the UML classifier RecordId. Defined as an unsigned long long it is capable to hold a 64 bit integer value, as required by the PIM.

15.3.9.2 Difference to the Telecom Log Service

The type RecordId is identical to the type used in the Telecom Log Service for simple log records.

15.3.10 LogRecord

```
struct LogRecord {
    RecordId          id;
    LogTime           time;
    ProducerLogRecord info;
};
```

```
typedef sequence<LogRecord> LogRecordSequence;
```

The **LogRecord** type defines the format of the log records as stored in the **Log**. The 'info' field is the **ProducerLogRecord** that is written by a producer client to the Log.

The **LogRecordSequence** type defines an unbounded sequence of **LogRecords**..

Struct member	Description
Id	This field uniquely identifies a log record in the Log.
Time	This field holds the timestamp for the record.
Info	This field contains the logging record supplied by the producer.

15.4 Logging Interfaces

Operations on the Log object are separated into three distinct concrete interfaces. Each of these interfaces represents a different access kind or privilege. This represents a lightweight method of protection for the underlying Log object, without adding any additional code. For the typically severe resource constrained embedded environments this Lightweight Logging Service is addressing, the code saving is important, and the protection functionality is considered sufficient.

15.4.1 Interface LogStatus

```
interface LogStatus {
   unsigned long long get_max_size();
   unsigned long long get_current_size();
   unsigned long long get_n_records();
   LogFullAction get_log_full_action();
   AvailabilityStatus get_availability_status();
   AdministrativeState get_administrative_state();
   OperationalState get_operational_state();
};
```

The purpose of this interface is to make common operations equally available in the three concrete interfaces inherited from this interface. These operations provide a common and consistent way to query the actual state of a **Log** object. No state changes are permitted or implied through the operations offered in this interface.

From a client's perspective, this interface should be considered as abstract; its operations should be invoked only in the context of the inherited interfaces.

15.4.1.1 get_max_size

Returns the size of the logging storage area.

Parameters and Return

Parameter	Type	Description
`<return>`	`unsigned long long`	The maximum size of the log storage area in bytes.

Exceptions

This function raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The available space in this storage area is finite. This operation returns the maximum capacity in bytes of the storage area.

15.4.1.2 get_current_size

Returns the amount of log storage area currently occupied by logging records.

Parameters and Return

Parameter	Type	Description
`<return>`	**`unsigned long long`**	The size of the currently used log storage area in bytes.

Exceptions

This function raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The **`get_current_size`** operation returns the size in bytes of the log storage area currently occupied by logging records. This value is less or equal to the total storage area size returned by the **`get_max_size`** operation.

15.4.1.3 get_n_records

Returns the number of records presently stored in the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	**`unsigned long long`**	The number of logging records currently stored in the storage area.

Exceptions

This operation raises no exceptions.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The **`get_n_records`** operation returns the number of logging records currently stored in the log storage area.

15.4.1.4 get_log_full_action

Returns the action taken when the storage area becomes full.

Parameters and Return

Parameter	Type	Description
`<return>`	`LogFullAction`	The actually selected alternative of the LogFullAction enumeration.

Exceptions

This operation raises no exceptions.

Description

Since the storage space of the Log storage area is finite, the Logging Service has to take special action when the free space is depleted. The kind of action is described by the **LogFullAction** type. The `get_log_full_action` operation returns the information about which action the Logging Service will take when the storage area becomes full. The possible values are **HALT**, which means no further logging records are accepted and stored; or **WRAP**, which means the **Log** continues by overwriting the oldest records in the storage area.

15.4.1.5 get_availability_status

Returns the availability status of the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	`AvailabilityStatus`	An instance of the AvailabilityStatus representing the actual status of the log.

Exceptions

This operation raises no exceptions.

Description

The ability of the Log to accept and store logging records might become impaired. The `get_availability_status` operation is used to check the availability status of the Log. The returned instance of the **AvailibilityStatus** type contains two Boolean values: **off_duty**, which indicates the log is disabled when true; and **log_full**, which indicates that all free space is depleted in the log storage area.

15.4.1.6 get_administrative_state

Returns the administrative state of the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	`AdministrativeState`	The actually selected alternative of the AdministrativeState enumeration.

Exceptions

This operation raises no exceptions.

Description

The ability of the logging service to accept and store new logging records can be affected by administrative action. The **`get_administrative_state`** is used to read the administrative state of the Log. The possible states are **locked** and **unlocked**. If the state is **locked**, no new records are accepted. Reading of already stored records is not affected.

15.4.1.7 get_operational_state

Returns the operational state of the Log.

Parameters and Return

Parameter	Type	Description
`<return>`	`OperationalState`	The actually selected alternative of the OperationalState enumeration.

Exceptions

This operation raises no exceptions.

Description

The **`get_operational_state`** operation returns the actual operational state of the log. Possible values are **enabled**, which means the log is fully functional and available to log producer and log consumer clients; or **disabled**, which indicates the log has encountered a runtime problem and is not available for use by log producers or log consumers.

15.4.2 Interface LogConsumer

```
interface LogConsumer : LogStatus {
   RecordId get_record_id_from_time (in LogTime fromTime);
   LogRecordSequence retrieve_records(inout RecordId currentId,
                           inout unsigned long howMany);
         LogRecordSequence retrieve_records_by_level(
                           inout RecordId currentId,
                           inout unsigned long howMany,
                           in LogLevelSequence valueList);
   LogRecordSequence retrieve_records_by_producer_id(
                           inout RecordId currentId,
                           inout unsigned long howMany,
                           in StringSeq valueList);
   LogRecordSequence retrieve_records_by_producer_name(
                           inout RecordId currentId,
                           inout unsigned long howMany,
                           in StringSeq valueList);
};
```

15.4.2.1 get_record_id_from_time

Identify a record in the log record based on its time stamp.

Parameters and Return

Parameter	Type	Description
`fromTime`	**`LogTime`**	The timestamp with which to start the search.
`<return>`	**`RecordId`**	Record ID of the first record matching the timestamp.

Exceptions

This operation raises no exceptions.

Description

The **`get_record_id_from_time`** operation returns the record Id of the first record in the Log with a time stamp that is greater than, or equal to, the time specified in the **`fromTime`** parameter. If the Log does not contain a record that meets the criteria provided, then the **RecordId** returned corresponds to the next record that will be recorded in the future. In this way, if this “future” **`recordId`** is passed into a retrieval operation, an empty record will be returned unless records have been recorded since the time specified. Note that if the time specified in the **`fromTime`** parameter is in the future, there is no guarantee that the resulting records returned by a retrieval operation will have a time stamp after the **`fromTime`** parameter if the returned **`recordId`** from this invocation of the **`get_record_id_from_time`** operation is subsequently used as input to the **`retrieveById`** operation.

15.4.2.2 retrieve_records

Retrieves a specified number of records from the Log.

Parameters and Return

Parameter	Type	Description
`currentId`	`RecordId`	The ID of the starting record.
`howMany`	`Unsigned long`	The number of records to retrieve.
`<return>`	`LogRecordSequence`	The sequence of retrieved records.

Exceptions

This operation raises no exceptions.

Description

The `retrieve_records` operation returns a **LogRecordSequence** that begins with the record specified by the `currentId` parameter. The number of records in the **LogRecordSequence** returned by the `retrieve_records` operation is equal to the number of records specified by the `howMany` parameter, or the number of records available if the number of records specified by the `howMany` parameter cannot be met. The log will update `howMany` to indicate the number of records returned and will set `currentId` to either the id of the record following the last examined record or the next record that will be recorded in the future if there are no further records available. If the record specified by `currentId` does not exist, but corresponds to the next record that will be recorded in the future, the `retrieve_records` operation returns an empty list of **LogRecords**, sets `howMany` to zero, and leaves the value of `currentId` unchanged. If the record specified by currentId does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, the `retrieve_records` operation returns an empty list of **LogRecords**, and sets both, `currentId` and `howMany` to zero. Note that this operation does not guarantee a return of sequential records in Log and modifies the `currentId` value. Consequently, subsequent invocation of this operation with the `get_record_id_from_time` operation may result in the Log consumer not being able to obtain some of the records.

15.4.2.3 retrieve_records_by_level

Retrieves a specified number of records from the Log that correspond to the provided log levels.

Parameters and Return

Parameter	Type	Description
`currentId`	`RecordId`	The ID of the starting record.
`howMany`	`Unsigned long`	The number of records to retrieve.
`valueList`	`LogLevelSequence`	The sequence of log levels that will be sought.
`<return>`	`LogRecordSequence`	The sequence of retrieved records.

Exceptions

This operation raises no exceptions.

Description

The **retrieve_records_by_level** operation returns a **LogRecordSequence** of records that correspond to the supplied **LogLevels**. Candidate records for the **LogRecordSequence** begin with the record specified by the **currentId** parameter. The number of records in the **LogRecordSequence** returned by the **retrieve_records_by_level** operation is equal to the number of records specified by the **howMany** parameter, or the number of records available if the number of records specified by the **howMany** parameter cannot be met. The log will update **howMany** to indicate the number of records returned and will set **currentId** to either the id of the record following the last examined record or the next record that will be recorded in the future if there are no further records available. If the record specified by **currentId** does not exist, but corresponds to the next record that will be recorded in the future, the **retrieve_records_by_level** operation returns an empty list of **LogRecords**, sets **howMany** to zero, and leaves the value of **currentId** unchanged. If the record specified by currentId does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, the **retrieve_records_by_level** operation returns an empty list of **LogRecords**, and sets both, **currentId** and **howMany** to zero. Note that this operation does not guarantee a return of sequential records in Log and modifies the **currentId** value. Consequently, subsequent invocation of this operation with the **get_record_id_from_time** operation may result in the Log consumer not being able to obtain some of the records.

15.4.2.4 retrieve_records_by_producer_id

Retrieves a specified number of records from the Log that correspond to the provided producer IDs.

Parameters and Return

Parameter	Type	Description
currentId	**RecordId**	The ID of the starting record.
howMany	**Unsigned long**	The number of records to retrieve.
valueList	**StringSeq**	The sequence of producer ids that will be sought.
<return>	**LogRecordSequence**	The sequence of retrieved records.

Exceptions

This operation raises no exceptions.

Description

The **retrieve_records_by_producer_id** operation returns a **LogRecordSequence** of records that correspond to the supplied **producerIds**. Candidate records for the **LogRecordSequence** begin with the record specified by the **currentId** parameter. The number of records in the **LogRecordSequence** returned by the **retrieve_records_by_producer_id** operation is equal to the number of records specified by the **howMany** parameter, or the number of records available if the number of records specified by the **howMany** parameter cannot be met. The log will update **howMany** to indicate the number of records returned and will set **currentId** to either the id of the record following the last examined record or the next record that will be recorded in the future if there are no further records available. If the record specified by **currentId** does not exist, but corresponds to the next record that will be recorded in the future, the **retrieve_records_by_producer_id** operation returns an empty list of **LogRecords**, sets **howMany** to zero, and leaves the value of **currentId** unchanged. If the record specified by currentId does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, the **retrieve_records_by_producer_id** operation returns an empty list of **LogRecords**, and sets both, **currentId** and **howMany** to zero. Note that this operation does not guarantee a return of sequential records in Log and modifies the **currentId** value. Consequently, subsequent invocation of this operation with the **get_record_id_from_time** operation may result in the Log consumer not being able to obtain some of the records.

retrieve_records_by_producer_name

Retrieves a specified number of records from the Log that correspond to the provided producer names.

Parameters and Return

Parameter	Type	Description
`currentId`	`RecordId`	The ID of the starting record.
`howMany`	`Unsigned long`	The number of records to retrieve.
`valueList`	`StringSeq`	The sequence of producer names that will be sought.
`<return>`	`LogRecordSequence`	The sequence of retrieved records.

Exceptions

This operation raises no exceptions.

Description

The **retrieve_records_by_producer_name** operation returns a **LogRecordSequence** of records that correspond to the supplied **producerNames.** Candidate records for the **LogRecordSequence** begin with the record specified by the **currentId** parameter. The number of records in the **LogRecordSequence** returned by the **retrieve_records_by_producer_name** operation is equal to the number of records specified by the **howMany** parameter, or the number of records available if the number of records specified by the **howMany** parameter cannot be met. The log will update **howMany** to indicate the number of records returned and will set **currentId** to either the id of the record following the last examined record or the next record that will be recorded in the future if there are no further records available. If the record specified by **currentId** does not exist, but corresponds to the next record that will be recorded in the future, the **retrieve_records_by_producer_name** operation returns an empty list of **LogRecords**, sets **howMany** to zero, and leaves the value of **currentId** unchanged. If the record specified by currentId does not exist and does not correspond to the next record that will be recorded in the future, or if the Log is empty, the **retrieve_records_by_producer_name** operation returns an empty list of **LogRecords**, and sets both, **currentId** and **howMany** to zero. Note that this operation does not guarantee a return of sequential records in Log and modifies the **currentId** value. Consequently, subsequent invocation of this operation with the **get_record_id_from_time** operation may result in the Log consumer not being able to obtain some of the records.

15.4.3 Interface LogProducer

```
interface LogProducer : LogStatus {
   oneway void write_records(
      in ProducerLogRecordSequence records);
   oneway void write_record(
      in ProducerLogRecord record);
};
```

This interface allows the insertion of new log records into the logging storage area encapsulated by the Log class. In favor of preserving the overall operational integrity of the system, no guarantee is made that a logging record is accepted and stored if the logging service is unable to process and /or store it.

15.4.3.1 write_records

Writes records to the Log.

Parameters and Return

Parameter	**Type**	**Description**
`records`	`ProducerLogRecordSequence`	The records to be written to the log.
`<return>`	`void`	This operation provides no return.

Exceptions

This operation raises no exceptions.

Description

The **`write_records`** operation adds the log records supplied in the **`records`** parameter to the Log. When there is insufficient storage to add one of the supplied log records to the Log, and the **LogFullAction** is set to **HALT**, the **`write_records`** operation will set the availability status logFull state to true. For example, if 3 records are provided in the records parameter, and while trying to write the second record to the log, the record will not fit, then the log is considered to be full. Therefore, the second and third records will not be stored in the log but the first record would have been successfully stored. When there is insufficient storage to add one of the supplied log records to the Log, and the **LogFullAction** is set to **WRAP**, the **`write_records`** operation will overwrite the oldest LogRecords with the newest records, as they are written to the Log, and leave the availability status logFull state unchanged.

The **`write_records`** operation inserts the current UTC time to the **`time`** field of each record written to the Log, and assigns a unique record id to the **`id`** field of the **LogRecord**.

Log records accepted for storage by the **`write_records`** will be available for retrieval in the order received.

Note – The purpose of the oneway invocation is, within the limitations of embedded ORBs, to de-couple the log producer from the logging service implementation, so that difficulties in the Log have no side-effects on the log producer or its operation. However, since ORBs may legally discard oneway requests, implementers should take extra care that the oneway invocations of **`write_records`** are not discarded without very substantial reason.

15.4.3.2 write_record

Writes a single record to the Log.

Parameters and Return

Parameter	Type	Description
`record`	`ProducerLogRecord`	The record to be written to the log.
`<return>`	`void`	This operation provides no return.

Exceptions

This operation raises no exceptions.

Description

The **`write_record`** operation adds a log record supplied in the **`record`** parameter to the Log. When there is insufficient storage to add the supplied log record to the Log, and the **LogFullAction** is set to **HALT**, the **`write_record`** operation will set the availability status logFull state to true. When there is insufficient storage to add the supplied log record to the Log, and the **LogFullAction** is set to **WRAP**, the **`write_record`** operation will overwrite the oldest LogRecords with the new record, and leave the availability status logFull state unchanged.

The **`write_record`** operation inserts the current UTC time to the **`time`** field of each record written to the Log, and assigns a unique record id to the **`id`** field of the **LogRecord**.

Log records accepted for storage by **`write_record`** will be available for retrieval in the order received.

Note – The purpose of the oneway invocation is, within the limitations of embedded ORBs, to de-couple the log producer from the logging service implementation, so that difficulties in the Log have no side-effects on the log producer or its operation. However, since ORBs may legally discard oneway requests, implementers should take extra care that the oneway invocations of **`write_record`** are not discarded without very substantial reason.

15.4.4 Interface LogAdministrator

```
interface LogAdministrator : LogStatus {
      void set_max_size(in unsigned long long size)
                raises (InvalidParam);
      void set_log_full_action(in LogFullAction action);
      void set_administrative_state(in AdministrativeState state);
      void clear_log();
      void destroy ();
};
```

This interface allows the retrieval of logging records from the storage area encapsulated by the Log class.

15.4.4.1 set_max_size

Sets the maximum size the Log storage area.

Parameters and Return

Parameter	Type	Description
`size`	`unsigned long long`	The desired size for the logging storage area in bytes.
`<return>`	`void`	This operation does not return a value.

Exceptions

This operation raises the InvalidParam exception if the supplied parameter is invalid.

Description

Logging records are stored in a storage area encapsulated by the **Log** class. The available space in this storage area is finite. This operation allows setting of the maximum capacity in bytes of the storage area. Note, however, that this operation might be constrained by the underlying operation (you can't assign more memory than is physically present), or a platform specific implementation might decide to render this operation as a no-op and provide a fixed maximum size instead.

15.4.4.2 set_log_full_action

Configure the action to be taken if the log storage area becomes full.

Parameters and Return

Parameter	Type	Description
`action`	`LogFullAction`	Specify the desired selection from the LogFullAction enumeration (either HALT or WRAP).
`<return>`	`void`	This operation does not return a value.

Exceptions

This operation raises no exceptions.

Description

Since the storage space of the Log storage area is finite, the Logging Service has to take special action when the free space is depleted. The kind of action is described by the **LogFullAction** type. The **`set_log_full_action`** operation allows the specification which action should be taken after all free space in the log storage area is depleted. The possible values are **HALT**, which means no further logging records are accepted and stored; or **WRAP**, which means the **Log** continues by overwriting the oldest records in the storage area. When the **LogFullAction** type is set to **WRAP**, the Log will set the availability status logFull state to false.

15.4.4.3 set_administrative_state

The **`set_administrative_state`** operation provides write access to the administrative state value.

Parameters and Return

Parameter	Type	Description
`state`	`unsigned long long`	Select the desired alternative from the AdministrativeState enumeration. (Possible values are **locked** and **unlocked**).
`<return>`	`void`	This operation does not return a value.

Exceptions

This operation raises no exceptions.

Description

This operation allows one to affect the ability of the logging service to accept and store new logging records by administrative action. The possible states are **locked** and **unlocked**. If the state is **locked**, no new records are accepted. Reading of already stored records is not affected. If the state is set to **unlocked**, the log operates normally.

15.4.4.4 clear_log

Purge the log storage area.

Parameters and Return

This operation has no parameters or returns.

Exceptions

This operation raises no exceptions.

Description

This operation purges all logging records from the log storage area; however, it does not alter the size of the storage area in any way. The log will set the availability status logFull state to false.

15.4.4.5 destroy

Tear down an instantiated Log.

Parameters and Return

This operation has no parameters or returns.

Exceptions

This operation raises no exceptions.

Description

This operation will destroy the associated instance of the **Log** class. All existing records in the log storage area are irrecoverably lost and the memory resources associated with the storage area are released.

15.5 Consolidated IDL

15.5.1 CosLwLog::LogStatus

```
#ifndef COS_LW_LOG_STATUS_IDL
#define COS_LW_LOG_STATUS_IDL

#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"
#endif

#if defined(CORBA_E_COMPACT)

module CosLwLog {

#ifndef _PRE_3_0_COMPILER_
  typeprefix CosLwLog "omg.org";
#endif

  // The following constants are intended to identify
  // the nature of a logging record. The constants
  // represent the valid values for LogLevel
  // The list of constants may be expanded
  const unsigned short SECURITY_ALARM = 1;
  const unsigned short FAILURE_ALARM = 2;
  const unsigned short DEGRADED_ALARM =3;
  const unsigned short EXCEPTION_ERROR =4;
  const unsigned short FLOW_CONTROL_ERROR =5;
  const unsigned short RANGE_ERROR =6;
  const unsigned short USAGE_ERROR = 7;
  const unsigned short ADMINISTRATIVE_EVENT = 8;
  const unsigned short STATISTIC_REPORT = 9;
  // Values ranging from 10 to 26 are reserved for
  // 16 debugging levels.
  typedef unsigned short LogLevel;

  enum OperationalState {disabled, enabled};
  enum AdministrativeState {locked, unlocked};
  enum LogFullAction {WRAP, HALT};

  typedef unsigned long long RecordId;

  struct LogTime {
    long seconds;
    long nanoseconds;
  };

  struct AvailabilityStatus{
    boolean off_duty;
    boolean log_full;
  };

  struct ProducerLogRecord {
    string producerId;
    string producerName;
    LogLevel level;
    string logData;
  };

  struct LogRecord {
    RecordId id;
    LogTime time;
    ProducerLogRecord info;
  };

  typedef sequence<LogRecord> LogRecordSequence;
  typedef sequence<ProducerLogRecord> ProducerLogRecordSequence;
  typedef sequence<LogLevel> LogLevelSequence;
  typedef sequence<string> StringSeq;

  exception InvalidParam {
    string details;
  };

  interface LogStatus {
    unsigned long long get_max_size();
    unsigned long long get_current_size();
    unsigned long long get_n_records();
    LogFullAction get_log_full_action();
    AvailabilityStatus get_availability_status();
    AdministrativeState get_administrative_state();
    OperationalState get_operational_state();
  };

};

#endif // defined(CORBA_E_COMPACT)
#endif // COS_LW_LOG_STATUS_IDL
```

15.5.2 CosLwLog::LogConsumer

```
#ifndef COS_LW_LOG_CONSUMER_IDL
#define COS_LW_LOG_CONSUMER_IDL

#if defined(CORBA_E_COMPACT)

#include <CosLwLogStatus.idl>

#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"
```

```
#endif

module CosLwLog {

  interface LogConsumer : LogStatus {
    RecordId get_record_id_from_time (in LogTime fromTime);
    LogRecordSequence retrieve_records(
        inout RecordId currentId,
        inout unsigned long howMany);
    LogRecordSequence retrieve_records_by_level(
        inout RecordId currentId,
        inout unsigned long howMany,
        in LogLevelSequence valueList);
    LogRecordSequence retrieve_records_by_producer_id(
        inout RecordId currentId,
        inout unsigned long howMany,
        in StringSeq valueList);
    LogRecordSequence retrieve_records_by_producer_name(
        inout RecordId currentId,
        inout unsigned long howMany,
        in StringSeq valueList);
   };

};

#endif // defined(CORBA_E_COMPACT)
#endif // COS_LW_LOG_CONSUMER_IDL
```

15.5.3 CosLwLog::LogProducer

```
#ifndef COS_LW_LOG_CONSUMER_IDL
#define COS_LW_LOG_CONSUMER_IDL

#if defined(CORBA_E_COMPACT)

#include <CosLwLogStatus.idl>

#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"
#endif

module CosLwLog {

  interface LogConsumer : LogStatus {
    RecordId get_record_id_from_time (in LogTime fromTime);
    LogRecordSequence retrieve_records(
        inout RecordId currentId,
        inout unsigned long howMany);
    LogRecordSequence retrieve_records_by_level(
        inout RecordId currentId,
        inout unsigned long howMany,
        in LogLevelSequence valueList);
    LogRecordSequence retrieve_records_by_producer_id(
        inout RecordId currentId,
        inout unsigned long howMany,
        in StringSeq valueList);
    LogRecordSequence retrieve_records_by_producer_name(
        inout RecordId currentId,
        inout unsigned long howMany,
        in StringSeq valueList);
   };

};

#endif // defined(CORBA_E_COMPACT)
#endif // COS_LW_LOG_CONSUMER_IDL
```

15.5.4 CosLwLog::LogAdministrator

```
#ifndef COS_LW_LOG_ADMINISTRATOR_IDL
#define COS_LW_LOG_ADMINISTRATOR_IDL

#if defined(CORBA_E_COMPACT)

#include <CosLwLogStatus.idl>

#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"
#endif

module CosLwLog {

  interface LogAdministrator : LogStatus {
    void set_max_size(in unsigned long long size) raises (InvalidParam);
    void set_log_full_action(in LogFullAction action);
    void set_administrative_state(in AdministrativeState state);
    void clear_log();
    void destroy ();
  };

};

#endif // defined(CORBA_E_COMPACT)
#endif // COS_LW_LOG_ADMINISTRATOR_IDL
```

15.5.5 CosLwLog::Log

```
#ifndef COS_LW_LOG_SERVICE_IDL
#define COS_LW_LOG_SERVICE_IDL
```

```
#if defined(CORBA_E_COMPACT)

#include <CosLwLogAdministrator.idl>
#include <CosLwLogConsumer.idl>
#include <CosLwLogProducer.idl>

#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"
#endif

module CosLwLog
{
  interface Log : LogAdministrator, LogConsumer, LogProducer {};
};

#endif // defined(CORBA_E_COMPACT)
#endif // COS_LW_LOG_SERVICE_IDL
```

16 General Inter-ORB Protocol

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter.

16.1 Overview

This chapter specifies a General Inter-ORB Protocol (GIOP) for ORB interoperability, which can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions.

16.2 Goals of the General Inter-ORB Protocol

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

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

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

16.3 GIOP Overview

The GIOP specification consists of the following elements:

- *The Common Data Representation (CDR) definition*. CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for “on-the-wire” transfer between ORBs and Inter-ORB bridges (agents).

- *GIOP Message Formats*. GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.
- *GIOP Transport Assumptions*. The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

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

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

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

The complete OMG IDL specifications for the GIOP and IIOP are shown in Section 11.5, “Consolidated IDL,” on page 19. Fragments of the specification are used throughout this chapter as necessary.

16.3.1 Common Data Representation (CDR)

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

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

16.3.2 GIOP Message Overview

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

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

CORBA::Object::get_interface).

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

16.3.3 GIOP Message Transfer

The GIOP specification is designed to operate over any connection-oriented transport protocol that meets a minimal set of assumptions (described in Section 16.4, "GIOP Message Transport," on page 277). GIOP uses underlying transport connections in the following ways:

- ***Asymmetrical connection usage*** - The GIOP defines two distinct roles with respect to connections, client, and server. The client side of a connection originates the connection, and sends object requests over the connection. In GIOP versions 1.0 and 1.1, the server side receives requests and sends replies. The server side of a connection may not send object requests. This restriction, which was included to make GIOP 1.0 and 1.1 much simpler and avoid certain race conditions, has been relaxed for GIOP version 1.2 and later, as specified in the BiDirectional GIOP specification.
- ***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.
- ***GIOP versions for requests and replies*** - The GIOP version of the message carrying a response to a request shall be the same as the GIOP version of the message carrying the request. This rule does not apply when the server is responding with a MessageError because it does not support the GIOP minor version in the request.

16.4 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 Section 16.4.1, “Connection Management,” on page 278 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.

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

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

Only GIOP messages are sent over GIOP connections.

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

16.4.1.1 Connection Closure

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

For GIOP versions 1.0, and 1.1:

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

disconnect has occurred, and treat the condition as an error.

For GIOP Version 1.2 and later:

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

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

- If there are any pending non-oneway requests, which were initiated on a connection by the ORB shutting down that connection, the connection-peer ORB should consider them as canceled.
- If an ORB receives a **CloseConnection** message from its connection-peer ORB, it should assume that any outstanding messages (i.e., without replies) were received after the connection-peer ORB sent the CloseConnection message, were not processed, and may be safely re-sent on a new connection.
- After issuing a **CloseConnection** message, the issuing ORB may close the connection. Some transport protocols (not including TCP) do not provide an “orderly disconnect” capability, guaranteeing reliable delivery of the last message sent. When GIOP is used with such protocols, an additional handshake needs to be provided as part of the mapping to that protocol’s connection mechanisms, to guarantee that both ends of the connection understand the disposition of any outstanding GIOP requests.

16.4.1.2 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 reusing connections, if it wishes. If not, the client may open a new connection for each active object supported by the server, although this behavior should be avoided.

16.4.2 Message Ordering

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

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

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

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

17 CDR Transfer Syntax

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

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter except Section 17.6.4, "Context," on page 303.

Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter except

- Section 17.5, "Value Types," on page 290,
- Section 17.6.1, "TypeCode," on page 297,
- Section 17.6.2, "Any," on page 303, and
- Section 17.6.4, "Context," on page 303.

17.1 Overview

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

An octet stream is an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine over some IPC mechanism or network transport. For the purposes of this discussion, an octet stream is an arbitrarily long (but finite) sequence of eight-bit values (octets) with a well-defined beginning. The octets in the stream are numbered from *0* to *n-1*, where *n* is the size of the stream. The numeric position of an octet in the stream is called its *index*. Octet indices are used to calculate alignment boundaries, as described in Section 17.2.1, "Alignment," on page 282.

GIOP defines two distinct kinds of octet streams:

- Message - an octet stream constituting the basic unit of information exchange in GIOP, described in detail in Section 15.2, "GIOP Message Formats," on page 2.
- Encapsulation - an octet stream into which OMG IDL data structures may be marshaled independently, apart from any particular message context, described in detail in Section 17.4, "Encapsulation," on page 289.

17.2 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see Section 15.2, "GIOP Message Formats," on page 2) include tags in message headers that indicate the byte ordering in the message. Encapsulations include an initial flag that indicates the byte ordering within the encapsulation, described in Section 17.4, "Encapsulation," on page 289. 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.

17.2.1 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 17.1 gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 17.1- Alignment requirements for OMG IDL primitive data types

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

Alignment is defined above as being relative to the beginning of an octet stream. The first octet of the stream is octet index zero (0); any data type may be stored starting at this index. Such octet streams begin at the start of a GIOP message header (see Section 15.2.1, “GIOP Message Header,” on page 2) and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See Section 17.4, “Encapsulation,” on page 289).

17.2.2 Integer Data Types

Figure 17.1 on page -283 illustrates the representations for OMG IDL integer data types, including the following data types:

- **short**

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

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

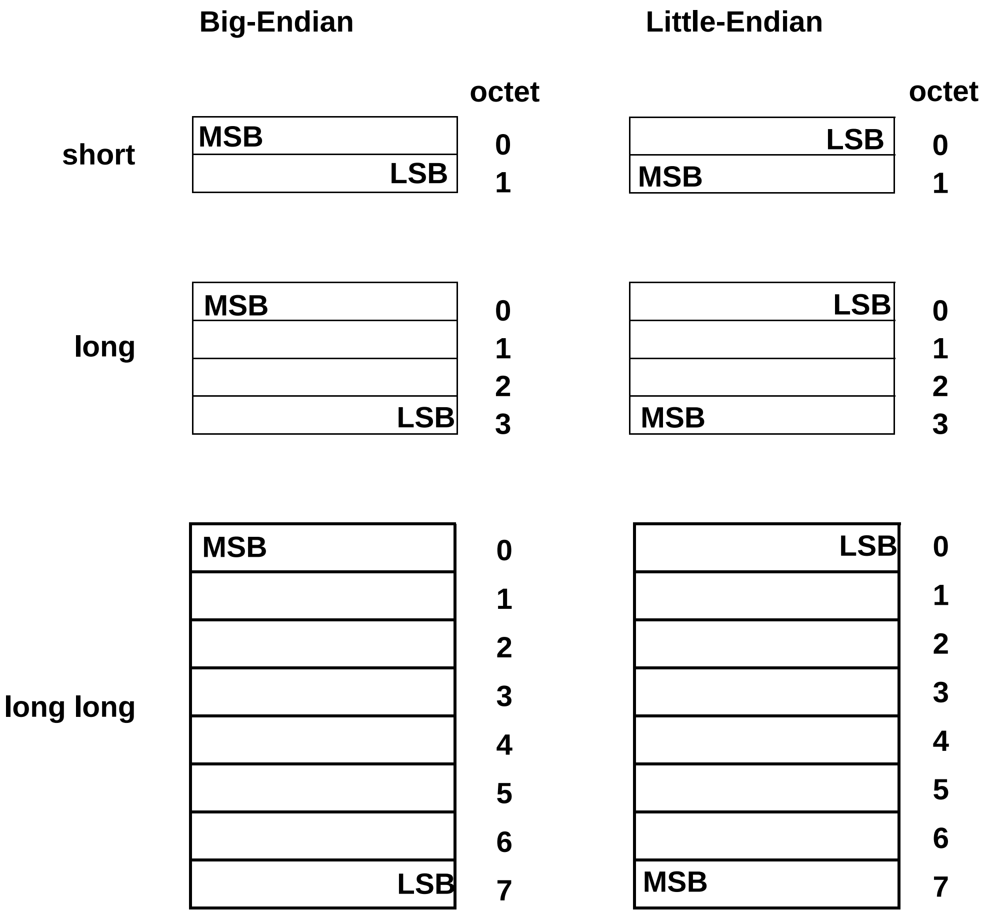

Figure 17.1Sizes and bit ordering in big-endian and little-endian encodings of OMG IDL integer data types, both signed and unsigned.

17.2.3 Floating Point Data Types

Figure 17.2 on page -285 illustrates the representation of floating point numbers. These exactly follow the IEEE standard formats for floating point numbers[1], selected parts of which are abstracted here for explanatory purposes. The diagram shows three different components for floating points numbers, the sign bit (s), the exponent (e) and the fractional part (f) of the mantissa. The sign bit has values of 0 or 1, representing positive and negative numbers, respectively.

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

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

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

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

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

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

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

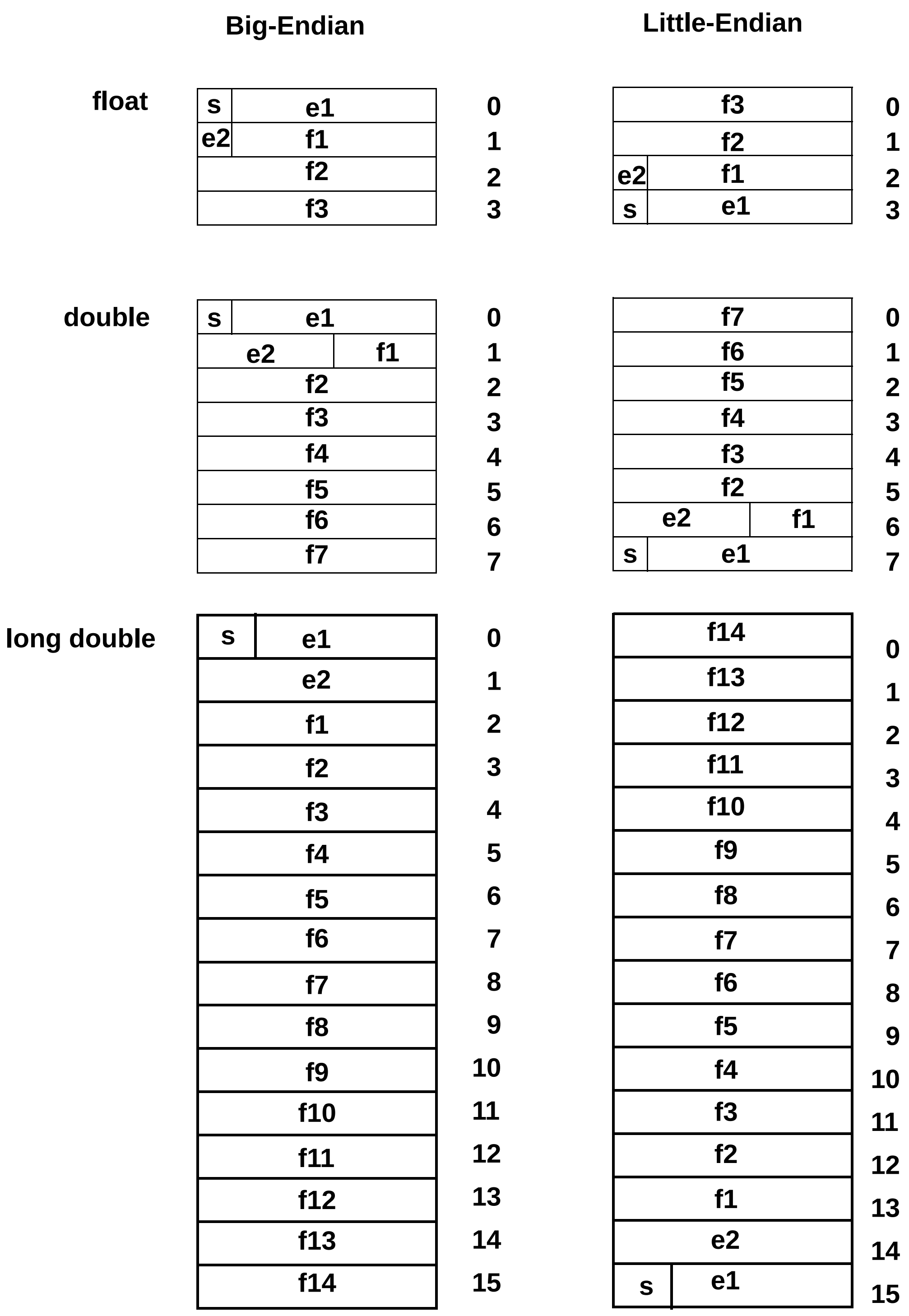

Figure 17.2- Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single, double precision, and double extended floating point numbers.

17.2.4 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.

17.2.5 Boolean

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

17.2.6 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. 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) 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 to determine the maximum length (max_bytes) of any character codepoint.

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

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

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

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

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

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

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

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

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

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

For GIOP 1.1, 1.2 and 1.3, UCS-2 and UCS-4 should be encoded using the endianess of the GIOP message, for backward compatibility.

For GIOP 1.4, the byte order rules for UCS-2 and UCS-4 are the same as for UTF-16.

UTF-16LE and UTF-16BE, from IANA codeset registry, have their own endianess definition. Thus these should be encoded using the endianess specified by their endianness definition.

17.3 OMG IDL Constructed Types

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

17.3.1 Alignment

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

17.3.2 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.

17.3.3 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.

17.3.4 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.

17.3.5 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.

17.3.6 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.

17.3.7 Strings and Wide Strings

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

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

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

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

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

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

17.3.8 Fixed-Point Decimal Type

The IDL **fixed** type has no alignment restrictions, and is represented as shown in Table 17.4 on page -289. Each **octet** contains (up to) two decimal digits. If the **fixed** type has an odd number of decimal digits, then the representation begins with the first (most significant) digit — d0 in the figure. Otherwise, this first half-octet is all zero, and the first digit is in the second half-octet — d1 in the figure. The sign configuration, in the last half-octet of the representation, is 0xD for negative numbers and 0xC for positive and zero values.

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

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

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

Figure 17.3- Decimal Digit Encoding for Fixed Type

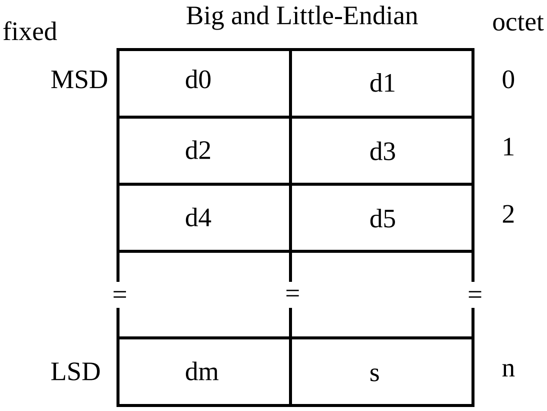

Figure 17.4- IDL Fixed Type Representation

17.4 Encapsulation

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.

The GIOP and IIOP explicitly use encapsulations in three places: *TypeCodes* (see Section 17.6.1, “TypeCode,” on page 297), the IIOP protocol profile inside an IOR (see Section 17.7, “Object References,” on page 303), and in service-specific context. In addition, some ORBs may choose to use an encapsulation to hold the **object_key**, or in other places that a **sequence<octet>** data type is in use.

When encapsulating OMG IDL data types, the first octet in the stream (index 0) contains a boolean value indicating the byte ordering of the encapsulated data. If the value is **FALSE** (0), the encapsulated data is encoded in big-endian order; if **TRUE** (1), the data is encoded in little-endian order, exactly like the byte order flag in GIOP message headers (see Section 15.2, "GIOP Message Formats," on page 2). This value is not part of the data being encapsulated, but is part of the octet stream holding the encapsulation. Following the byte order flag, the data to be encapsulated is marshaled into the buffer as defined by CDR encoding rules. Marshaled data are aligned relative to the beginning of the octet stream (the first octet of which is occupied by the byte order flag).

When the encapsulation is encoded as type **sequence<octet>** for subsequent marshaling, an unsigned long value containing the sequence length is prefixed to the octet stream, as prescribed for sequences (see Section 17.3.5, "Sequence," on page 288). The length value is not part of the encapsulation's octet stream, and does not affect alignment of data within the encapsulation.

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

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

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

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

17.5 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.

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

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.

17.5.1 Partial Type Information and Versioning

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

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

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

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

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

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

For value types marshaled as abstract interfaces, **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.

17.5.2 Example

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

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

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

1. If there is a single operation:

op1(in animal a);

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

valuetype horse: animal ...

then the encoding is as shown below:

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

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

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

valuetype horse: truncatable animal ...

then the encoding is as shown below

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

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

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

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

2. Given the additional operation:

   op2(in horse h);

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

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

   valuetype horse: animal ...

   then the encoding is as shown below:

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

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

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

valuetype horse: truncatable animal ...

then the encoding is as shown below:

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

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

17.5.3 Scope of the Indirections

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

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

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

17.5.4 Null Values

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

17.5.5 Other Encoding Information

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

17.5.6 Fragmentation

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

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

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

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

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

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

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

- End tags, chunk size tags, and value tags are encoded using non-overlapping ranges so that the unmarshaling code can tell after reading each chunk whether:
    - another chunk follows (positive tag).
    - one or multiple value types are ending at a given point in the stream (negative tag).
    - a nested value follows (special large positive tag).
- The end tag is a negative long whose value is the negation of the absolute nesting depth of the value type ending at this point in the CDR stream. Any value types that have not already been ended and whose nesting depth is greater than the depth indicated by the end tag are also implicitly ended. The end tag value **0** is reserved for future use (e.g., supporting a nesting depth of more than **2^31**). The outermost value type will always be terminated by an end tag with a value of **-1**. Enclosing non-chunked valuetypes are not considered when determining the nesting depth.

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

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

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

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

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

- For the purposes of chunking, values encoded as indirections or null are treated as non-value data.
- The scope of an encoded valuetype is a complete GIOP message or an encapsulation. Starting a new encapsulation starts a new scope. Ending an encapsulation ends the current scope and restores the previous scope. Starting a new scope starts a new count of end tag nesting (initially 0), chunking status (initially false) and chunk position (initially 0).
- Chunks in the same scope are never nested. When a value is nested within another value, the outer value's chunk ends at the place in the stream where the inner value starts. If the outer value has non-value data to be marshaled following the inner value, the end tag for the inner value is followed by a continuation chunk for the remainder of the outer value.
- Regardless of the above rules, any value nested within a chunked value in the same scope is always chunked. Furthermore, any such nested value that is truncatable must encode its type information as a list of RepositoryIDs (see Section 17.5.1, “Partial Type Information and Versioning,” on page 291).

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.

17.5.7 Notation

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

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

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

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

17.5.8 The Format

```
(1)                <value>  ::= <value_tag> [  <codebase_URL> ]
                                [ <type_info> ] <state>
|                                                        <value_ref>
(2)            <value_ref>  ::= <indirection_tag>  <indirection> | <null_tag>
(3)            <value_tag>  ::= long                     // 0x7fffff00 <= value_tag <= 0x7fffffff
(4)            <type_info>  ::= <rep_ids> | <repository_id>
(5)                <state>  ::= <octets> |<value_data>* [ <end_tag> ]
(6)           <value_data>  ::= <value_chunk> | <value>
(7)              <rep_ids>  ::= long <repository_id>+
|                                                        <indirection_tag> <indirection>
(8)        <repository_id>  ::= string | <indirection_tag> <indirection>
(9)          <value_chunk>  ::=                          <chunk_size_tag> <octets>
(10)            <null_tag>  ::= (long) 0
```

```
(11)    <indirection_tag>  ::=  (long) 0xffffffff
(12)     <codebase_URL>    ::=  string | <indirection_tag> <indirection>
(13)   <chunk_size_tag>    ::=  long
                                //  0 < chunk_size_tag < 2^31-256 (0x7fffff00)
(14)            <end_tag>  ::=  long                      // -2^31 < end_tag < 0
(15)        <indirection>  ::=  long // -2^31 < indirection < 0
(16)             <octets>  ::=  octet | octet <octets>
```

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

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

17.6 Pseudo-Object Types

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

17.6.1 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 17.1 on page -298. 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.

17.6.1.1 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 17.1. A "simple" parameter list has a fixed number of fixed length entries, or a single parameter that has its length encoded first.
- All other typecodes have *complex parameter lists*, which are encoded as the **TCKind** enum value followed by a CDR encapsulation octet sequence (see Section 17.4, "Encapsulation," on page 289) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 17.1.

The third column of Table 17.1 shows whether each parameter list is *empty*, *simple*, or *complex*. Also, note that an internal indirection facility is needed to represent some kinds of typecodes; this is explained in "Indirection: Recursive and Repeated TypeCodes" on page 302. This indirection does not need to be exposed to application programmers.

17.6.1.2 TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 17.1 on page -298. The ordering and meaning of parameters is a superset of those given in Section 5.5, "TypeCodes," on page 17; more information is needed by CDR's representation in order to provide the full semantics of **TypeCodes** as shown by the API.

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

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

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

Table 17.1- TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
tk_null	0	empty	– none –
tk_void	1	empty	– none –
tk_short	2	empty	– none –

Table 17.1- TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
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)}
tk_enum	17	complex	string (repository ID), string (name), ulong (count) {string (member name)}
tk_string	18	simple	ulong (max length[b])

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

Table 17.1- TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
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)}
tk_value_box	30	complex	string (repository ID), string(name), TypeCode
tk_native	31	complex	string (repository ID), string(name)
tk_abstract_interface	32	complex	string(RepositoryId), string(name)
tk_local_interface	33	complex	string(RepositoryId), string(name)
tk_component	34	complex	string (repository ID), string(name)

Table 17.1- TypeCode enum values, parameter list types, and parameters

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

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. Should be tk_null if there is no concrete base.

f. See “Indirection: Recursive and Repeated TypeCodes” on page 302.

Encoded Identifiers and Names

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

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

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

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.

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

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

Encoding the tk_union Default Case

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

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

TypeCodes for Multi-Dimensional Arrays

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

Indirection: Recursive and Repeated TypeCodes

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

CDR provides a constrained indirection to resolve this problem:

- For GIOP 1.2 and below, 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.
- For GIOP 1.3 and above, the indirection applies only to TypeCodes nested within some "top-level" TypeCode, or from one top-level TypeCode to another. Indirected TypeCodes nested within a top-level TypeCode can only reference TypeCodes that are part of the same top-level TypeCode, including the top-level TypeCode itself. Indirected top-level TypeCodes can reference other top-level TypeCodes but cannot reference TypeCodes nested within some other top-level TypeCode.
- Only the second (and subsequent) references to a **TypeCode** in that scope may use the indirection facility. The first reference to that **TypeCode** must be encoded using the normal rules. In the case of a recursive **TypeCode**, this means that the first instance will not have been fully encoded before a second one must be completely encoded.

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

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

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

17.6.2 Any

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

17.6.3 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.

17.6.4 Context

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

17.6.5 Exception

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

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

17.7 Object References

Object references are encoded in OMG IDL. 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.

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.

18 GIOP Messages

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter.

18.1 Overview

This chapter specifies the message formats and protocol used in the General Inter-ORB Protocol (GIOP) for ORB interoperability.

18.2 GIOP Message Formats

GIOP has restriction on client and server roles with respect to initiating and receiving messages. For the purpose of GIOP versions 1.0 and 1.1, a client is the agent that opens a connection (see more details in Section 13.4.1, “Connection Management,” on page 5) and originates requests. Likewise, for GIOP versions 1.0 and 1.1, a server is an agent that accepts connections and receives requests.When Bidirectional GIOP is in use for GIOP protocol version 1.2 and 1.3, either side may originate messages.

GIOP message types are summarized in Table 18.1, which lists the message type names, whether the message is originated by client, server, or both, and the value used to identify the message type in GIOP message headers.

Table 18.1- GIOP Message Types and Originators

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

Table 18.1- GIOP Message Types and Originators

Message Type	Originator	Value	GIOP Versions
CloseConnection	Server	5	1.0, 1.1, 1.2, 1.3
CloseConnection	Both	5	1.2, 1.3
MessageError	Both	6	1.0, 1.1, 1.2, 1.3
Fragment	Both	7	1.1, 1.2, 1.3

18.2.1 GIOP Message Header

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

```
module GIOP { // IDL extended for version 1.1, 1.2, and 1.3
    struct Version {
        octet         major;
        octet         minor;
    };
#if MAX_GIOP_VERSION_NUMBER == 0
    // 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  // MAX_GIOP_VERSION_NUMBER

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

    // GIOP 1.1
    struct MessageHeader_1_1 {
        Magicn              magic;
        Version             GIOP_version;
        octet               flags;          // GIOP 1.1 change
        octet               message_type;
        unsigned long       message_size;
    };

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

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

- **magic** identifies GIOP messages. The value of this member is always the four (upper case) characters “GIOP,” encoded in ISO Latin-1 (8859.1).
- **GIOP_version** contains the version number of the GIOP protocol being used in the message. The version number applies to the transport-independent elements of this specification (i.e., the CDR and message formats) that constitute the GIOP. This is not equivalent to the IIOP version number (as described in Section 14.7, “Object References,” on page 27) though it has the same structure. The major GIOP version number of this specification is one (1); the minor versions are zero (0), one (1), and two (2).

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

  A client should not send a GIOP message having a higher minor version number than that published by the server in the tag Internet IIOP Profile body of an IOR.
- **byte_order** (in GIOP 1.0 only) indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of **FALSE** (0) indicates big-endian byte ordering, and **TRUE** (1) indicates little-endian byte ordering.
- flags (in GIOP 1.1, 1.2, and 1.3) is an 8-bit octet. The least significant bit indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of **FALSE** (0) indicates big-endian byte ordering, and **TRUE** (1) indicates little-endian byte ordering. The byte order for fragment messages must match the byte order of the initial message that the fragment extends.

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

  The most significant 6 bits are reserved. These 6 bits must have value 0 for GIOP version 1.1, 1.2, and 1.3.
- **message_type** indicates the type of the message, according to Table 18.1; these correspond to enum values of type **MsgType**.
- **message_size** contains the number of octets in the message following the message header, encoded using the byte order specified in the byte order bit (the least significant bit) in the **flags** field (or using the byte_order field in GIOP 1.0). It refers to the size of the message body, not including the 12-byte message header. This count includes any alignment gaps and must match the size of the actual request parameters (plus any final padding bytes that may follow the parameters to have a fragment message terminate on an 8-byte boundary).

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

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

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

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

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

18.2.2 Request Message

Request messages encode CORBA object invocations, including attribute accessor operations, and **CORBA::Object** operations **get_interface, repository_id,** 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

18.2.2.1 Request Header

The request header is specified as follows:

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

    // GIOP 1.0
    struct RequestHeader_1_0 { // Renamed from RequestHeader
        IOP::ServiceContextList        service_context;
        unsigned long                  request_id;
        boolean                        response_expected;
        IOP::ObjectKey                 object_key;
        string                         operation;
        CORBA::OctetSeq                requesting_principal;
    };
    typedef octet RequestReserved[3];
    struct RequestHeader_1_1 {
        IOP::ServiceContextList        service_context;
        unsigned long                  request_id;
        boolean                        response_expected;
        RequestReserved                reserved; // Added in GIOP 1.1
        IOP::ObjectKey                  object_key;
        string                          operation;
        CORBA::OctetSeq                 requesting_principal;
    };

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

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

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

    struct RequestHeader_1_2 {
        unsigned long                   request_id;
        octet                           response_flags;
        RequestReserved                 reserved; // Added in GIOP 1.1
        TargetAddress                   target;
        string                          operation;
        IOP::ServiceContextList         service_context;
        // requesting_principal not in GIOP 1.2 and 1.3
    };
    typedef RequestHeader_1_2 RequestHeader_1_3;
};
```

The members have the following definitions:

- **request_id** is used to associate reply messages with request messages (including **LocateRequest** messages). The client (requester) is responsible for generating values so that ambiguity is eliminated; specifically, a client must not re-use **request_id** values during a connection if: *(a)* the previous request containing that ID is still pending, or *(b)* if the previous request containing that ID was canceled and no reply was received. (See the semantics of the Section 18.2.4, “CancelRequest Message,” on page 314).
- **response_flags** is set to 0x0 for a **SyncScope** of **NONE** and **WITH_TRANSPORT**. The flag is set to 0x1 for a **SyncScope** of **WITH_SERVER**. A non exception reply to a request message containing a **response_flags** value of 0x1 should contain an empty body, i.e., the equivalent of a void operation with no out/inout parameters. The flag is set to 0x3 for a **SyncScope** of **WITH_TARGET**. These values ensure interworking compatibility between this and previous versions of **GIOP**.

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

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

  In the case of **CORBA::Object** operations that are defined in the CORBA Core and that correspond to GIOP request messages, the operation names are **_interface**, **_is_a**, **_non_existent, _domain_managers, _component** and **_repository_id**.

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

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

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

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

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

18.2.2.2 Request Body

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

- All **in** and **inout** parameters, in the order in which they are specified in the operation's OMG IDL definition, from left to right.
- An optional **Context** pseudo object. 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
};
```

18.2.3 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

18.2.3.1 Reply Header

The reply header is defined as follows:

```
module GIOP {                           // IDL extended for 1.2 and 1.3
#if MAX_GIOP_MINOR_VERSION < 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
#endif
#if MAX_GIOP_MINOR_VERSION >= 2

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

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

};
```

The members have the following definitions:

- **request_id** is used to associate replies with requests. It contains the same **request_id** value as the corresponding request.
- **reply_status** indicates the completion status of the associated request, and also determines part of the reply body contents. If no exception occurred and the operation completed successfully, the value is NO_EXCEPTION and the body contains return values. Otherwise the body
    - contains an exception, or
    - directs the client to reissue the request to an object at some other location, or
    - directs the client to supply more addressing information.
- **service_context** contains ORB service data being passed from the server to the client, encoded as described in Section 13.3.3, “GIOP Message Transfer,” on page 4.

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

18.2.3.2 Reply Body

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

- If the **reply_status** value is NO_EXCEPTION, the body is encoded as if it were a structure holding first any operation return value, then any **inout** and **out** parameters in the order in which they appear in the operation’s OMG

IDL definition, from left to right. (That structure could be empty.)

- If the **reply_status** value is USER_EXCEPTION or SYSTEM_EXCEPTION, then the body contains the exception that was raised by the operation. (Only the user-defined exceptions listed in the operation's OMG IDL definition may be raised.) When a GIOP Reply message contains a `**reply_status**' value of SYSTEM_EXCEPTION, the body of the Reply message conforms to the following structure:

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

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

Note – OMG standard minor codes are identified with the 20 bit **VMCID \x4f4d0**. This appears as the characters 'O' followed by the character 'M' on the wire**,** which is defined as a 32-bit constant called **OMGVMCID \x4f4d0000** (see Section 5.6.4, "Standard Minor Exception Codes," on page 33) so that allocated minor code numbers can be or-ed with it to obtain the **minor_code_value**.

- If the **reply_status** value is **LOCATION_FORWARD**, then the body contains an object reference (IOR) encoded as described in Section 14.7, "Object References," on page 27. The client ORB is responsible for re-sending the original request to that (different) object. This resending is transparent to the client program making the request.
- The usage of the **reply_status** value **LOCATION_FORWARD_PERM** behaves like the usage of **LOCATION_FORWARD**, but when used by a server it also provides an indication to the client that it may replace the old IOR with the new IOR. Both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.
- If the **reply_status** value is **NEEDS_ADDRESSING_MODE** then the body contains a **GIOP::AddressingDisposition**. The client ORB is responsible for re-sending the original request using the requested addressing mode. The resending is transparent to the client program making the request.

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

For example, the reply body for a successful response (the value of **reply_status** is **NO_EXCEPTION**) to the *Request* example shown on page -311 would be equivalent to the following structure:

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

Note that the **object_key** field in any specific GIOP profile is server-relative, not absolute. Specifically, when a new object reference is received in a **LOCATION_FORWARD Reply** or in a **LocateReply** message, the **object_key** field embedded in the new object reference's GIOP profile may not have the same value as the **object_key** in the GIOP profile of the original object reference. For details on location forwarding, see Section 18.3, "Object Location," on page 319.

18.2.4 CancelRequest Message

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

CancelRequest messages have two elements, encoded in this order:

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

18.2.4.1 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.

18.2.5 LocateRequest Message

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

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

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

LocateRequest messages have two elements, encoded in this order:

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

18.2.5.1 LocateRequest Header.

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

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

// GIOP 1.0
    struct LocateRequestHeader_1_0 {
                                    // Renamed LocationRequestHeader
        unsigned long                   request_id;
        IOP::ObjectKey                  object_key;
    };

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

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

The members are defined as follows:

- **request_id** is used to associate **LocateReply** messages with **LocateRequest** ones. The client (requester) is responsible for generating values; see Section 18.2.2, “Request Message,” on page 308 for the applicable rules.
- For GIOP 1.0 and 1.1, **object_key** identifies the object being located. In an IIOP context, this value is obtained from the **object_key** field from the encapsulated **IIOP::ProfileBody** in the IIOP profile of the IOR for the target object. When GIOP is mapped to other transports, their IOR profiles must also contain an appropriate corresponding value. This value is only meaningful to the server and is not interpreted or modified by the client.
- For GIOP 1.2$_{, 1.3}$, target identifies the object being located. The possible values of this union are:
    - **KeyAddr** is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.
    - **ProfileAddr** is the transport-specific GIOP profile selected for the target’s IOR by the client ORB.
    - **IORAddressingInfo** is the full IOR of the target object. The **selected_profile_index** indicates the transport-specific GIOP profile that was selected by the client ORB.

See Section 18.3, “Object Location,” on page 319 for details on the use of **LocateRequest**.

18.2.6 LocateReply Message

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

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

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

18.2.6.1 Locate Reply Header

The locate reply header is defined as follows:

```
module GIOP {                           // IDL extended for GIOP 1.2 and 1.3
#if MAX_GIOP_MINOR_VERSION < 2
    // GIOP 1.0 and 1.1
    enum LocateStatusType_1_0 {// Renamed from LocateStatusType
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

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

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

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

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

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.

18.2.6.2 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.**

LocateReply bodies are marshaled immediately following the LocateReply header.

18.2.7 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 or later both sides of the connection may send the **CloseConnection** message.

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

For details on the usage of **CloseConnection** messages, see Section 13.4.1, “Connection Management,” on page 5.

18.2.8 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.

18.2.9 Fragment Message

This message is added in GIOP 1.1.

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

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

In GIOP 1.1, a **Request** or **Reply** message can be broken into multiple fragments. In GIOP 1.2 and later, a **Request**, **Reply**, **LocateRequest**, or **LocateReply** message can be broken into multiple fragment. The first fragment is a regular message (e.g., **Request** or **Reply**) with the **more** fragments bit in the **flags** field set to **TRUE**. This initial fragment can be followed by one or more messages using the fragment messages. The last fragment shall have the more fragment bit in the flag field set to **FALSE**.

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

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

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

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

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

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

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

```
module GIOP {//IDL extension for GIOP 1.2 and later

    struct FragmentHeader_1_2 {
        unsigned long request_id;
    };
    typedef FragmentHeader_1_2 FragmentHeader_1_3;
};
```

18.3 Object Location

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

A given transport address does not necessarily correspond to any specific ORB architectural component (such as an *object adapter, object server process, Inter-ORB bridge,* and so forth). It merely implies the existence of some agent with which a connection may be opened, and to which requests may be sent.

The "agent" (owner of the server side of a connection) may have one of the following roles with respect to a particular object reference:

- The agent may be able to accept object requests directly for the object and return replies. The agent may or may not own the actual object implementation; it may be an Inter-ORB bridge that transforms the request and passes it on to another process or ORB. From GIOP's perspective, it is only important that requests can be sent directly to the agent.
- The agent may not be able to accept direct requests for any objects, but acts instead as a location service. Any **Request** messages sent to the agent would result in either exceptions or replies with **LOCATION_FORWARD** status, providing new addresses to which requests may be sent. Such agents would also respond to **LocateRequest** messages with appropriate **LocateReply** messages.
- The agent may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.
- The agent may directly respond to requests for a particular object at one point in time, and provide a forwarding location at a later time (perhaps during the same connection).

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

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

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

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

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

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

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

18.4 Service Context

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

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

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

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

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

```
module IOP {                                        // IDL

    typedef unsigned long           ServiceId;
    typedef CORBA::OctetSeq ContextData;
    struct ServiceContext {
        ServiceId           context_id;
        ContextData         context_data;
    };
    typedef sequence <ServiceContext>ServiceContextList;
};
```

The context data for a particular service will be encoded as specified for its service-specific OMG IDL definition, and that encoded representation will be encapsulated in the **context_data** member of **IOP::ServiceContext**. (See Section 14.4, “Encapsulation,” on page 11). 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 24 bits of a service context ID contain a 24-bit vendor service context codeset ID (VSCID); the low-order 8 bits contain the rest of the service context ID. A vendor (or group of vendors) who wishes to define a specific set of service context IDs should obtain a unique VSCID from the OMG, and then define a specific set of service context IDs using the VSCID for the high-order bits.

The VSCIDs of zero to 15 inclusive (0x000000 to 0x00000f) are 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).

18.4.1 Standard Service Contexts

```
module IOP {                                            // IDL
    const ServiceId         TransactionService = 0;
    const ServiceId         CodeSets = 1;
    const ServiceId         ChainBypassCheck = 2;
    const ServiceId         ChainBypassInfo = 3;
    const ServiceId         LogicalThreadId = 4;
    const ServiceId         BI_DIR_IIOP = 5;
    const ServiceId         SendingContextRunTime = 6;
    const ServiceId         INVOCATION_POLICIES = 7;
    const ServiceId         FORWARDED_IDENTITY = 8;
    const ServiceId         UnknownExceptionInfo = 9;
    const ServiceId         RTCorbaPriority = 10;
    const ServiceId         RTCorbaPriorityRange = 11;
    const ServiceId         FT_GROUP_VERSION = 12;
    const ServiceId         FT_REQUEST = 13;
    const ServiceId         ExceptionDetailMessage = 14;
    const ServiceId         SecurityAttributeService = 15;
    const ServiceId         ActivityService = 16;
    const ServiceId         RMICustomMaxStreamForma = 17;
    const ServiceId         ACCESS_SESSION_ID = 18;
    const ServiceId         SERVICE_SESSION_ID = 19;
    const ServiceId         FIREWALL_PATH = 20;
    const ServiceId         FIREWALL_PATH_RESP = 21;
};
```

The standard **ServiceId**s currently defined are:

- **TransactionService** identifies a CDR encapsulation of the **CosTransactions::PropogationContext** defined in the Object Transaction Service specification (formal/00-06-28).
- **CodeSets** identifies a CDR encapsulation of the **CONV_FRAME::CodeSetContext**.
- DCOM-CORBA Interworking uses three service contexts as defined in "DCOM-CORBA Interworking" in the “Interoperability with non-CORBA Systems”chapter. They are:
  - **ChainBypassCheck**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassCheck**. This is carried only in a **Request** message.
  - **ChainBypassInfo**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassInfo**. This is carried only in a **Reply** message.
  - **LogicalThreadId**, which carries a CDR encapsulation of the **struct CosBridging::LogicalThreadId**.
- **BI_DIR_IIOP** identifies a CDR encapsulation of the **IIOP::BiDirIIOPServiceContext**.
- **SendingContextRunTime** identifies a CDR encapsulation of the IOR of the **SendingContext::RunTime** object.
- For information on **INVOCATION_POLICIES** refer to the CORBA Messaging chapter.
- For information on **FORWARDED_IDENTITY** refer to the Firewall specification (orbos/98-05-04).
- **UnknownExceptionInfo** identifies a CDR encapsulation of a marshaled instance of a **java.lang.throwable** or one of its subclasses as described in Java to IDL Language Mapping, “Mapping of UnknownExceptionInfo Service Context,” section.
- For information on **RTCorbaPriority** refer to the Real-time CORBA specification.
- For information on **RTCorbaPriorityRange** refer to the Real-time CORBA specification.
- **FT_GROUP_VERSION, FT_REQUEST** - refer to the Fault Tolerant CORBA chapter.
- **ExceptionDetailMessage** identifies a CDR encapsulation of a wstring, encoded using GIOP 1.2 with a TCS-W of UTF-16. This service context may be sent on Reply messages with a reply_status of SYSTEM_EXCEPTION or USER_EXCEPTION. The usage of this service context is defined by language mappings.
- **SecurityAttributeService** - refer to the Secure Interoperability chapter.
- **ActivityService** - refer to the Additional Structuring Mechanisms for OTS specification (orbos/01-11-08).
- **RMICustomMaxStreamFormat** - refer to Java to IDL Language Mapping.
- **ACCESS_SESSION_ID** and **SERVICE_SESSION_ID** - refer to the TSAS Specification.
- **FIREWALL_PATH** and **FIREWALL_PATH_RESP** - refer to the Firewall Service v2.0 Specification.

18.4.2 Service Context Processing Rules

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

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

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

18.5 Propagation of Messaging QoS

This section defines the profile Component through which QoS requirements are expressed in an object reference, and the Service Context through which QoS requirements are expressed as part of a GIOP request.

```
module Messaging {
    typedef CORBA::OctetSeq PolicyData;
    struct PolicyValue {
        CORBA::PolicyType           ptype;
        PolicyData                  pvalue;
    };
    typedef sequence<PolicyValue> PolicyValueSeq;

    const IOP::ComponentId TAG_POLICIES =           2;
    const IOP::ServiceId INVOCATION_POLICIES =      7;
};
```

18.5.1 Structures

PolicyValue

This structure contains the value corresponding to a Policy of the **PolicyType** indicated by its **ptype**. This representation allows the compact transmission of QoS policies within IORs and Service Contexts. The format of **pvalue** for each type is given in the specification of that Policy.

18.5.2 Messaging QoS Profile Component

A new **IOP::TaggedComponent** is defined for transmission of QoS policies within interoperable Object References. The body of this Component is a CDR encapsulation containing a **Messaging::PolicyValueSeq**. When creating Object references, Portable Object Adapters may encode the relevant policies with which it was created in this **TaggedComponent**. POA Policies that are exported in this way are clearly noted as *client-exposed* in their definitions. These policies are reconciled with the effective client-side override when clients invokes operations on that reference. For example, if a POA is created with a **RequestPriorityPolicy** with minimum value 0 and maximum value 10, all Object references created by that POA will have that default **RequestPriorityPolicy** encoded in their IOR. Furthermore, if a client sets an overriding **RequestPriorityPolicy** with both minimum and maximum of 5 (the client requires its requests to have a priority of value 5), the ORB will reconcile the effective Policy for any invocations on this Object reference to have a priority of 5 (since this value is within the range of priorities allowed by the target). On the other hand, if the client set an override with minimum value of 11, any invocation attempts would raise the system exception INV_POLICY.

18.5.3 Messaging QoS Service Context

A new **IOP::ServiceContext** is defined for transmission of QoS policies within GIOP requests and replies. The body of this Context is a CDR encapsulation containing a **Messaging::PolicyValueSeq**.

18.6 Consolidated IDL

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

    struct Version {
        octet       major;
        octet       minor;
    };

#if MAX_GIOP_MINOR_VERSION == 0
    // GIOP 1.0
    enum MsgType_1_0{   // rename from MsgType
        Request, Reply,     CancelRequest,
        LocateRequest,      LocateReply,
        CloseConnection,    MessageError
        };

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

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

    // GIOP 1.1
    struct MessageHeader_1_1 {
        Magicn                  magic;
        Version                 GIOP_version;
        octet                   flags;          // GIOP 1.1 change
        octet                   message_type;
        unsigned long           message_size;
    };
    // GIOP 1.2 and later
    typedef MessageHeader_1_1 MessageHeader_1_2;
    typedef MessageHeader_1_1 MessageHeader_1_3;

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

    // GIOP 1.1
    typedef octet RequestReserved[3];
    struct RequestHeader_1_1 {
        IOP::ServiceContextList   service_context;
        unsigned long             request_id;
        boolean                   response_expected;
        IOP::ObjectKey            object_key;
        string                    operation;
        CORBA::OctetSeq           requesting_principal;
    };

    // GIOP 1.2, and later
    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:             IOP::ObjectKey object_key;
        case ProfileAddr:         IOP::TaggedProfile profile;
        case ReferenceAddr:       IORAddressingInfo ior;
    };
    struct RequestHeader_1_2 {
        unsigned long             request_id;
        octet                     response_flags;
        RequestReserved           reserved; // Added in GIOP 1.1
        TargetAddress             target;
        string                    operation;
        // requesting_principal not in GIOP 1.2 and later
        IOP::ServiceContextList   service_context;   // 1.2 change
    };
#if MAX_GIOP_MINOR_VERSION < 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

#endif // MAX_GIOP_VERSION_NUMBER

#if MAX_GIOP_MINOR_VERSION >= 2

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

    struct ReplyHeader_1_2 {
        unsigned long                  request_id;
        ReplyStatusType_1_2            reply_status;
        IOP::ServiceContextList        service_context;    // 1.2 change
    };
    typedef ReplyHeader_1_2 ReplyHeader_1_3;
#endif // MAX_GIOP_VERSION_NUMBER
    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;
        IOP::ObjectKey                  object_key;
    };

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

    // GIOP 1.2 and later
        struct LocateRequestHeader_1_2 {
            unsigned long                   request_id;
            TargetAddress                   target;
    };
    typedef LocateRequestHeader_1_2 LocateRequestHeader_1_3;

#if MAX_GIOP_MINOR_VERSION < 2

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

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

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

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

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

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

module Messaging {
    //
    // Propagation of QoS Policies
    //

    typedef CORBA::OctetSeq PolicyData;
    struct PolicyValue {
        CORBA::PolicyType               ptype;
        PolicyData                      pvalue;
    };
    typedef sequence<PolicyValue> PolicyValueSeq;

};
```

19 Internet Interoperability Protocol (IIOP)

Compliance

Conformant implementations of the CORBA/*e* Compact Profile must comply with all clauses of this chapter. Conformant implementations of the CORBA/*e* Micro Profile must comply with all clauses of this chapter.

19.1 Overview

The original Interoperability RFP defines interoperability as the ability for a client on ORB A to invoke an OMG 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 implementation.
- 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.

19.2 Internet Inter-ORB Protocol (IIOP)

The baseline transport specified for GIOP is TCP/IP[1]. Specific APIs for libraries supporting TCP/IP may vary, so this discussion is limited to an abstract view of TCP/IP and management of its connections. The mapping of GIOP message transfer to TCP/IP connections is called the Internet Inter-ORB Protocol (IIOP).

IIOP 1.0 is based on GIOP 1.0.

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

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

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

IIOP 1.4 can be based on any of the GIOP minor versions 1.0, 1.1, 1.2$_{\text{, 1.3 or 1.4}}$. An IIOP 1.4 client must support GIOP 1.4, and may also support lesser GIOP minor versions. An IIOP 1.4 server must also support processing messages with all lesser GIOP versions.

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

Conformance to IIOP versions 1.1, 1.2, 1.3 and 1.4 requires support of Limited-Profile IOR conformance (see Section 19.3.2, “Interoperable Object References: IORs,” on page 334), specifically for the IIOP IOR Profile. As of CORBA 2.4, this limited IOR conformance is deprecated, and ORBs implementing IIOP are strongly recommended to support Full IOR conformance. Some future IIOP versions could require support of Full IOR conformance.

19.2.1 TCP/IP Connection Usage

Agents that are capable of accepting object requests or providing locations for objects (i.e., servers) publish TCP/IP addresses in IORs, as described in Section 19.4.1, “IIOP IOR Profiles,” on page 347. A TCP/IP address consists of an IP host address, typically represented by a host name, and a TCP port number. Servers must listen for connection requests.

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

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

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

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

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

19.2.2 IIOP IOR Profiles

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

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

    struct ProfileBody_1_0 {// renamed from ProfileBody
        Version                         iiop_version;
        string                          host;
        unsigned short                  port;
        IOP::ObjectKey                  object_key;
    };

    struct ProfileBody_1_1 {// also used for 1.2 and later
        Version                     iiop_version;
        string                      host;
        unsigned short              port;
        IOP::ObjectKey                  object_key;

    // Added in 1.1 unchanged for 1.2 and later
        IOP::TaggedComponentSeq components;
    };
};
```

IIOP Profile version number:

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

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

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

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

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

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

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

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

- **host** identifies the Internet host to which GIOP messages for the specified object may be sent. In order to promote a very large (Internet-wide) scope for the object reference, this will typically be the fully qualified domain name of the host, rather than an unqualified (or partially qualified) name. However, per Internet standards, the host string may also contain a host address expressed in standard "dotted decimal" form (e.g., "192.231.79.52").
- **port** contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.
- **object_key** is an opaque value supplied by the agent producing the IOR. This value will be used in request messages to identify the object to which the request is directed. An agent that generates an object key value must be able to map the value unambiguously onto the corresponding object when routing requests internally.
- **components** is a sequence of **TaggedComponent**, which contains additional information that may be used in making invocations on the object described by this profile. **TaggedComponent**s that apply to IIOP 1.2 are described below in Section 19.4.2, "IIOP IOR Profile Components," on page 349. Other components may be included to support enhanced versions of IIOP, to support ORB services such as security, and to support other GIOPs, ESIOPs, and proprietary protocols. If an implementation puts a non-standard component in an IOR, it cannot be assured that any or all non-standard components will remain in the IOR.

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

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

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

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

19.2.3 IIOP IOR Profile Components

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

- **TAG_ORB_TYPE**
- **TAG_CODE_SETS**
- **TAG_SEC_NAME**
- **TAG_ASSOCIATION_OPTIONS**
- **TAG_GENERIC_SEC_MECH**

- TAG_SSL_SEC_TRANS
- TAG_SPKM_1_SEC_MECH
- TAG_SPKM_2_SEC_MECH
- TAG_KerberosV5_SEC_MECH
- TAG_CSI_ECMA_Secret_SEC_MECH
- TAG_CSI_ECMA_Hybrid_SEC_MECH
- TAG_SSL_SEC_TRANS
- TAG_CSI_ECMA_Public_SEC_MECH
- TAG_FIREWALL_TRANS
- TAG_JAVA_CODEBASE
- TAG_TRANSACTION_POLICY
- TAG_MESSAGE_ROUTERS
- TAG_INET_SEC_TRANS

The following components are part of IIOP 1.2 (and later) conformance. All these components are optional.

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

19.3 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 *General Inter-ORB Protocol* chapter, *Object References* section.

19.3.1 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?* 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.

19.3.2 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.

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;

    typedef CORBA::OctetSeq ProfileData;
    struct TaggedProfile {
        ProfileId                   tag;
        ProfileData                 profile_data;
    };
    typedef sequence <TaggedProfile> TaggedProfileSeq ;

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

    struct IOR {
        string                      type_id;
        TaggedProfileSeq            profiles;
    };

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

    typedef unsigned long ComponentId;
    typedef CORBA::OctetSeq ComponentData;

    struct TaggedComponent {
        ComponentId              tag;
        ComponentData            component_data;
    };

    typedef sequence<TaggedComponent> TaggedComponentSeq;
    typedef sequence <TaggedComponent> MultipleComponentProfile;

    typedef CORBA::OctetSeq ObjectKey;
};
```

19.3.3 IOR Profiles

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 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.

When a specific protocol is used to convey an object reference passed as a parameter in an IDL operation invocation (or reply), an IOR which reflects, in its contained profiles, the full protocol understanding of the operation client (or server in case of reply) may be sent. A receiving ORB which operates (based on topology and policy information available to it) on profiles rather than the received IOR as a whole, to create a derived reference for use in its own domain of reference, is placing itself as a bridge between reference domains. Interoperability inhibiting situations can arise when an orb sends an IOR with multiple profiles (using one of its supported protocols) to a receiving orb, and that receiving orb later returns a derived reference to that object, which has had profiles or profile component data removed or transformed from the original IOR contents.

To assist in classifying behavior of ORBS in such bridging roles, two classes of IOR conformance may be associated with the conformance requirements for a given ORB interoperability protocol:

- Full IOR conformance requires that an orb which receives an IOR for an object passed to it through that ORB interoperability protocol, shall recover the original IOR, in its entirety, for passing as a reference to that object from that orb through that same protocol
- Limited-Profile IOR conformance requires that an orb which receives an IOR passed to it through a given ORB interoperability protocol, shall recover all of the standard information contained in the IOR profile for that protocol, whenever passing a reference to that object, using that same protocol, to another ORB.

Note – Conformance to IIOP versions 1.0, 1.1 and 1.2 only requires support of limited-Profile IOR conformance, specifically for the IIOP IOR profile. However, due to interoperability problems induced by Limited-Profile IOR conformance, it is now deprecated by the CORBA 2.4 specification for an orb to not support Full IOR conformance. Some future IIOP versions could require Full IOR conformance.

An ORB may be unable to use any of the profiles provided in an IOR for various reasons which may be broadly categorized as transient ones like temporary network outage, and non-transient ones like unavailability of appropriate protocol software in the ORB. The decision about the category of outage that causes an ORB to be unable to use any profile from an IOR is left up to the ORB. At an appropriate point, when an ORB discovers that it is unable to use any profile in an IOR, depending on whether it considers the reason transient or non-transient, it should raise the standard system exception TRANSIENT with standard minor code 2, or IMP_LIMIT with the standard minor code 1.

Each profile has a unique numeric tag, assigned by the OMG. The ones defined here are for the IIOP (see Section 19.4.2, "IIOP IOR Profile Components," on page 349) 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). 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.

ORBs claiming to support the Full-IOR conformance are required to preserve all the semantic content of any IOR (including the ordering of each profile and its components), and may only apply transformations which preserve semantics (e.g., changing Byte order for encapsulation).

For example, consider an echo operation for object references:

```
interface Echoer {Object echo(in Object o);};
```

Assume that the method body implementing this "echo" operation simply returns its argument. When a client application invokes the echo operation and passes an arbitrary object reference, if both the client and server ORBs claim support to Full IOR conformance, the reference returned by the operation is guaranteed to have not been semantically altered by either client or server ORB. That is, all its profiles will remain intact and in the same order as they were present when the reference was sent. This requirement for ORBs which claim support for Full-IOR conformance, ensures that, for example, a client can safely store an object reference in a naming service and get that reference back again later without losing information inside the reference.

19.3.4 Standard IOR Profiles

```
module IOP {
    const ProfileId             TAG_INTERNET_IOP = 0;
    const ProfileId             TAG_MULTIPLE_COMPONENTS = 1;
    const ProfileId             TAG_SCCP_IOP = 2;
    const ProfileId             TAG_UIPMC = 3;
};
```

19.3.4.1 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 Section 19.4.1, “IIOP IOR Profiles,” on page 347, 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.

The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in Section 19.4.1, “IIOP IOR Profiles,” on page 347.

19.3.4.2 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, the use of which must be specified by the protocol using this profile. This profile may be used to carry IOR components, as specified in Section 19.3.5, “IOR Components,” on page 337.

The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type shown above.

19.3.4.3 The TAG_SCCP_IOP Profile

See the CORBA/IN Interworking specification (dtc/2000-02-02).

19.3.4.4 The TAG_UIPMC Profile

The **TAG_UIPMC** tag is used by MIOP.

19.3.5 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.

Specifications of protocols must describe how the components affect the protocol. In addition, a protocol definition must specify, for each TaggedComponent, whether inclusion of the component in profiles supporting the protocol is required (MANDATORY PRESENCE) or not required (OPTIONAL PRESENCE). An ORB claiming to support Full-IOR conformance shall not drop optional components, once they have been added to a profile.

19.3.6 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;
    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_FIREWALL_TRANS = 23;
    const ComponentId    TAG_SCCP_CONTACT_INFO = 24;
    const ComponentId    TAG_JAVA_CODEBASE = 25;
    const ComponentId    TAG_TRANSACTION_POLICY = 26;
    const ComponentId    TAG_MESSAGE_ROUTERS = 30;
    const ComponentId    TAG_OTS_POLICY = 31;
    const ComponentId    TAG_INV_POLICY = 32;
    const ComponentId    TAG_CSI_SEC_MECH_LIST = 33;
    const ComponentId    TAG_NULL_TAG = 34;
    const ComponentId    TAG_SECIOP_SEC_TRANS = 35;
    const ComponentId    TAG_TLS_SEC_TRANS = 36;
    const ComponentId    TAG_ACTIVITY_POLICY = 37;
    const ComponentId    TAG_RMI_CUSTOM_MAX_STREAM_FORMAT = 38;
    const ComponentId    TAG_GROUP = 39;
    const ComponentId    TAG_GROUP_IIOP = 40;
    const ComponentId    TAG_PASSTHRU_TRANS = 41;
    const ComponentId    TAG_FIREWALL_PATH = 42;
    const ComponentId    TAG_IIOP_SEC_TRANS = 43;
    const ComponentId    TAG_INET_SEC_TRANS = 123;
};
```

The following additional components that can be used by other protocols are specified in the DCE ESIOP chapter of this document and *CORBAServices*, Security Service, in the Security Service for DCE ESIOP section:

```
    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
```

19.3.6.1 TAG_ORB_TYPE Component

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

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

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

19.3.6.2 TAG_ALTERNATE_IIOP_ADDRESS Component

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

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

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

encoded as a CDR encapsulation.

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

19.3.6.3 Other Components

The following standard components are specified in various OMG specifications:

- **TAG_CODE_SETS** - See CORBA Core specification (formal/04-03-01).
- **TAG_POLICIES** - See CORBA Messaging.

- **TAG_SEC_NAME** - See the Security Service specification, Mechanism Tags section.
- **TAG_ASSOCIATION_OPTIONS** - See the Security Service specification, Tag Association Options section.
- **TAG_SSL_SEC_TRANS** - See the Security Service specification, Mechanism Tags section.
- **TAG_GENERIC_SEC_MECH** and all other tags with names in the form **TAG_*_SEC_MECH** - See the Security Service specification, Mechanism Tags section.
- **TAG_FIREWALL_SEC** - See the Firewall specification (orbos/98-05-04).
- **TAG_SCCP_CONTACT_INFO** - See the CORBA/IN Interworking specification (telecom/98-10-03).
- **TAG_JAVA_CODEBASE** - See the Java to IDL Language Mapping specification (formal/99-07-59), Codebase Transmission section.
- **TAG_TRANSACTION_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_MESSAGE_ROUTERS** - See CORBA Messaging.
- **TAG_OTS_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_INV_POLICY** - See the Object Transaction Service specification (formal/00-06-28).
- **TAG_INET_SEC_TRANS** - See the Security Service specification (formal/00-06-25).
- **TAG_CSI_SEC_MECH_LIST**, **TAG_NULL_TAG**, **TAG_SECIOP_SEC_TRANS**, **TAG_TLS_SEC_TRANS** - See the Secure Interoperability chapter.
- **TAG_ACTIVITY_POLICY** - See the Additional Structuring Mechanisms for OTS specification (orbos/01-11-08).
- **TAG_RMI_CUSTOM_MAX_STREAM_FORMAT** - See Java to IDL Language Mapping Specification (formal/03-09-04).
- **TAG_GROUP** and **TAG_GROUP_IIOP** - Used in MIOP.
- **TAG_IIOP_SEC_TRANS** - Used in CSIv2. See CORBA Core specification (formal/04-03-01).
- **TAG_COMPLETE_OBJECT_KEY** - See CORBA Core specification (formal/04-03-01).
- **TAG_ENDPOINT_ID_POSITION** See CORBA Core specification (formal/04-03-01).
- **TAG_LOCATION_POLICY** - See CORBA Core specification (formal/04-03-01).
- **TAG_DCE_STRING_BINDING** - See CORBA Core specification (formal/04-03-01).
- **TAG_DCE_BINDING_NAME** - See CORBA Core specification (formal/04-03-01).
- **TAG_DCE_NO_PIPES** - See CORBA Core specification (formal/04-03-01).

19.3.7 Profile and Component Composition in IORs

The following rules augment the preceding discussion:

1. Profiles must be independent, complete, and self-contained. Their use shall not depend on information contained in

another profile.

2. Any invocation uses information from exactly one profile.
3. Information used to drive multiple inter-ORB protocols may coexist within a single profile, possibly with some information (e.g., components) shared between the protocols, as specified by the specific protocols.
4. Unless otherwise specified in the definition of a particular profile, multiple profiles with the same profile tag may be included in an IOR.
5. Unless otherwise specified in the definition of a particular component, multiple components with the same component tag may be part of a given profile within an IOR.
6. A **TAG_MULTIPLE_COMPONENTS** profile may hold components shared between multiple protocols. Multiple such profiles may exist in an IOR.
7. The definition of each protocol using a **TAG_MULTIPLE_COMPONENTS** profile must specify which components it uses, and how it uses them.
8. Profile and component definitions can be either public or private. Public definitions are those whose tag and data format is specified in OMG documents. For private definitions, only the tag is registered with OMG.
9. Public component definitions shall state whether or not they are intended for use by protocols other than the one(s) for which they were originally defined, and dependencies on other components.

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

19.3.8 IOR Creation and Scope

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

19.3.9 Stringified Object References

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

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

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

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

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

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

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

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

The hexadecimal strings are generated by first turning an object reference into an IOR, and then encapsulating the IOR using the encoding rules of CDR, as specified in GIOP 1.0. 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.

19.3.10 Object URLs

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

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

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

19.3.10.1 corbaloc URL

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

```
corbaloc::555xyz.com/Prod/TradingService
corbaloc:iiop:1.1@555xyz.com/Prod/TradingService
corbaloc::555xyz.com,:556xyz.com:80/Dev/NameService
corbaloc:rir:/TradingService
corbaloc:rir:/NameService
corbaloc:iiop:192.168.14.25:555/NameService
corbaloc::[1080::8:800:200C:417A]:88/DefaultEventChannel
```

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

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

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

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

The full syntax is:

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

<rir_prot_addr>       = <rir_prot_token>”:”
<rir_prot_token>      = “rir”

<iiop_prot_addr>      = <iiop_id><iiop_addr>
<iiop_id>             = “:” | <iiop_prot_token>”:”
<iiop_prot_token>      = “iiop”
<iiop_addr>           = [<version> <host> [“:” <port>]]
<host>                = DNS_style_host_name | IPv4_address
                          | "[" IPv6_address "]"
<version>             = <major> “.” <minor> “@” | empty_string
<port>                = number
<major>               = number
<minor>               = number

<future_prot_addr>    = <future_prot_id><future_prot_addr>
<future_prot_id>      = <future_prot_token>”:”
<future_prot_token>   = possible examples: “atm” | “dce”
<future_prot_addr>    = protocol specific address

<key_string>          = <string> | empty_string
```

Where:

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

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

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

rir_prot_addr: resolve_initial_references protocol identifier. This protocol does not have a version tag or address. See 19.3.9.2, ’corbaloc:rir URL’.

iiop_prot_addr: iiop protocol identifier, version tag, and address containing a DNS-style host name or IP address. See 19.3.9.3, ’corbaloc:iiop URL’” for the iiop specific definitions.

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

future_prot_id: token representing a protocol terminated with a “:”.

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

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

key_string: a stringified object key.

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

“;” | “/” | “:” | “?”| “:” | “@” | “&” | “=” | “+” | “$” |

“,” | “-” | “_” | “!” | “~” | “*” | “’” | “(“ | “)”

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

19.3.10.2 corbaloc:rir URL

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

The protocol address syntax is:

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

Where:

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

rir_prot_token: The token “**rir**” identifies this protocol.

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

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

19.3.10.3 corbaloc:iiop URL

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

```
<iiop_prot_addr>      = <iiop_id><iiop_addr>
<iiop_id>             = <iiop_default> | <iiop_prot_token>”:”
<iiop_default>        = “:”
<iiop_prot_token>     = “iiop”
<iiop_addr>           = [<version> <host> [“:” <port>]]

<host>                = DNS_style_host_name | IPv4_address
                            | "[" IPv6_address "]"
<version>             = <major> “.” <minor> “@” | empty_string
<port>                = number
<major>               = number
<minor>               = number
```

Where:

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

iiop_id: tokens recognized to indicate an iiop protocol corbaloc.

iiop_default: default token indicating iiop protocol, “**:**”.

iiop_prot_token: iiop protocol token, “**iiop**”

iiop_address: a single address

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

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

IPv4_address: numeric IPv4 address (dotted decimal notation)

IPv6_address: numeric IPv6 address (colon separated hexadecimal or mixed hexadecimal/decimal notation as described in RFC 2373)

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

19.3.10.4 corbaloc Server Implementation

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

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

19.3.10.5 corbaname URL

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

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

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

corbaname:rir:#a/local/obj

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

19.3.10.6 Future corbaloc URL Protocols

This specification only defines use of iiop with corbaloc. New protocols can be added to **corbaloc** as required. Each new protocol must implement the <future_prot_addr> component of the URL and define a described in Section 19.3.9.1, "corbaloc URL," on page 343."

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

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

19.3.10.7 Future URL Schemes

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

Table 19.1 lists the required and some optional formats.

Table 19.1 URL formats

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

19.4 IIOP IOR

19.4.1 IIOP IOR Profiles

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

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

    struct ProfileBody_1_0 {// renamed from ProfileBody
        Version                         iiop_version;
        string                          host;
        unsigned short                  port;
        IOP::ObjectKey                  object_key;
    };

    struct ProfileBody_1_1 {// also used for 1.2 and later
        Version                     iiop_version;
        string                      host;
        unsigned short              port;
        IOP::ObjectKey              object_key;

    // Added in 1.1 unchanged for 1.2 and later
        IOP::TaggedComponentSeq components;
    };
};
```

IIOP Profile version number:

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

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

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

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

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

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

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

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

- **host** identifies the Internet host to which GIOP messages for the specified object may be sent. In order to promote a very large (Internet-wide) scope for the object reference, this will typically be the fully qualified domain name of the host, rather than an unqualified (or partially qualified) name. However, per Internet standards, the host string may also contain a host address expressed in standard “dotted decimal” form (e.g., “192.231.79.52”).
- **port** contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.
- **object_key** is an opaque value supplied by the agent producing the IOR. This value will be used in request messages to identify the object to which the request is directed. An agent that generates an object key value must be able to map the value unambiguously onto the corresponding object when routing requests internally.
- **components** is a sequence of **TaggedComponent**, which contains additional information that may be used in making invocations on the object described by this profile. **TaggedComponent**s that apply to IIOP 1.2 are described below in Section 19.4.2, “IIOP IOR Profile Components,” on page 349. Other components may be included to support enhanced versions of IIOP, to support ORB services such as security, and to support other GIOPs, ESIOPs, and proprietary protocols. If an implementation puts a non-standard component in an IOR, it cannot be assured that any or all non-standard components will remain in the IOR.

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

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

- When there is a need to specify conformance rules that include the new components, there will be a need to create a new IIOP version.

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

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

19.4.2 IIOP IOR Profile Components

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

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

The following components are part of IIOP 1.2 (and later) conformance. All these components are optional.

- **TAG_ALTERNATE_IIOP_ADDRESS**
- **TAG_POLICIES**
- **TAG_DCE_STRING_BINDING**
- **TAG_DCE_BINDING_NAME**
- **TAG_DCE_NO_PIPES**
- T**AG_DCE_MECH**
- **TAG_COMPLETE_OBJECT_KEY**
- **TAG_ENDPOINT_ID_POSITION**
- **TAG_LOCATION_POLICY**
- **TAG_OTS_POLICY**

- **TAG_INV_POLICY**
- **TAG_CSI_SEC_MECH_LIST**
- **TAG_NULL_TAG**
- **TAG_SECIOP_SEC_TRANS**
- **TAG_TLS_SEC_TRANS**
- **TAG_ACTIVITY_POLICY**

19.5 Consolidated IDL

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

    struct ProfileBody_1_0 {// renamed from ProfileBody
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        IOP::ObjectKey          object_key;
    };

    struct ProfileBody_1_1 {// also used for 1.2, and later
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        IOP::ObjectKey          object_key;

        // Added in 1.1 unchanged for 1.2, and later
        IOP::TaggedComponentSeq components;
    };

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

    typedef sequence<ListenPoint> ListenPointList;
};
```

Annex A OMG IDL Tags and Exceptions

(normative)

This annex lists the standardized profile, service, component, policy tags and exception codes described in the CORBA documentation. Implementor-defined tags can also be registered in this manual. Requests to register tags with the OMG should be sent to *tag_request@omg.org*.

A.1 Profile ID Tags

Tag Name	Tag Value	Described in
ProfileId	TAG_INTERNET_IOP = 0	*ORB Interoperability Architecture* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ProfileId	TAG_MULTIPLE_COMPONENTS = 1	*ORB Interoperability Architecture* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ProfileId	TAG_SCCP_IOP = 2	CORBA/TC Interworking specification http://www.omg.org/technology/documents/formal/corba_tc_interworking_and_sccp_i.htm
ProfileId	TAG_UIPMC = 3	Unreliable Multicast (orbos/01-03-01)
ProfileId	TAG_MOBILE_TERMINAL_IOP = 4	Telecom Wireless specification - http://www.omg.org/technology/documents/formal/telecom_wireless.htm.

A.2 Service ID Tags

Tag Name	Tag Value	Described in
ServiceId	TransactionService = 0	Object Transaction Service specification http://www.omg.org/technology/documents/formal/transaction_service.htm
ServiceId	CodeSets = 1	*ORB Interoperability Architecture* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	ChainBypassCheck = 2	*Interoperability with non-CORBA Systems* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	ChainBypassInfo = 3	*Interoperability with non-CORBA Systems* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	LogicalThreadId = 4	*Interoperability with non-CORBA Systems* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01

Tag Name	Tag Value	Described in
ServiceId	BI_DIR_IIOP = 5	*General Inter-ORB Protocol* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	SendingContextRunTime = 6	*Value Type Semantics* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	INVOCATION_POLICIES = 7	*CORBA Messaging* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	FORWARDED_IDENTITY = 8	Firewall Traversal specification (ptc/04-03-01)
ServiceId	UnknownExceptionInfo = 9	Java to IDL Language Mapping specification: http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm
ServiceId	RTCorbaPriority = 10	Real-Time CORBA specification: see http://www.omg.org/technology/documents/formal/real-time_CORBA.htm
ServiceId	RTCorbaPriorityRange = 11	Real-Time CORBA specification: see http://www.omg.org/technology/documents/formal/real-time_CORBA.htm
ServiceId	FT_GROUP_VERSION = 12	*Fault Tolerant CORBA* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	FT_REQUEST= 13	*Fault Tolerant CORBA* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	ExceptionDetailMessage = 14	*Interoperability Architecture* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	SecurityAttributeService = 15	*Secure Interoperability* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ServiceId	ActivityService = 16	Additional Structuring Mechanisms for the OTS: http://www.omg.org/technology/documents/formal/add_struct.htm
ServiceId	RMICustomMaxStreamFormat = 17	Java to IDL Language Mapping specification: http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm
ServiceId	ACCESS_SESSION_ID = 18	Telecom Service Access Subscription (TSAS) specification: http://www.omg.org/technology/documents/formal/tsas.htm
ServiceId	SERVICE_SESSION_ID = 19	Telecom Service Access Subscription (TSAS) specification: http://www.omg.org/technology/documents/formal/tsas.htm
ServiceId	FIREWALL_PATH = 20	Firewall Traversal specification (ptc/04-03-01)
ServiceId	FIREWALL_PATH_RESP = 21	Firewall Traversal specification (ptc/04-03-01)

A.3 Component ID Tags

Tag Name	Tag Value	Described in
ComponentId	TAG_ORB_TYPE = 0	*Interoperability Architecture* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_CODE_SETS = 1	*Interoperability Architecture* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_POLICIES = 2	*CORBA Messaging* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_ALTERNATE_IIOP_ADDRESS = 3	*General Inter-ORB Protocol* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_COMPLETE_OBJECT_KEY = 5	*The DCE ESIOP* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_ENDPOINT_ID_POSITION = 6	*The DCE ESIOP* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_LOCATION_POLICY = 12	*The DCE ESIOP* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_ASSOCIATION_OPTIONS =13	Security Service specification: see http://www.omg.org/technology/documents/formal/security_service.htm
ComponentId	TAG_SEC_NAME = 14	
ComponentId	TAG_SPKM_1_SEC_MECH = 15	
ComponentId	TAG_SPKM_2_SEC_MECH = 16	
ComponentId	TAG_KerberosV5_SEC_MECH = 17	
ComponentId	TAG_CSI_ECMA_Secret_SEC_MECH = 18	
ComponentId	TAG_CSI_ECMA_Hybrid_SEC_MECH = 19	
ComponentId	TAG_SSL_SEC_TRANS = 20	
ComponentId	TAG_CSI_ECMA_Public_SEC_MECH = 21	
ComponentId	TAG_GENERIC_SEC_MECH = 22	
ComponentId	TAG_FIREWALL_TRANS = 23	Firewall Traversal specification (ptc/04-03-01)
ComponentId	TAG_SCCP_CONTACT_INFO = 24	CORBA/TC Interworking specification http://www.omg.org/technology/documents/formal/corba_tc_interworking_and_sccp_i.htm
ComponentId	TAG_JAVA_CODEBASE = 25	Java to IDL Language Mapping specification: http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm
ComponentId	TAG_TRANSACTION_POLICY = 26	Object Transaction Service specification http://www.omg.org/technology/documents/formal/transaction_service.htm
ComponentId	TAG_ FT_GROUP= 27	*Fault Tolerant CORBA* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_ FT_PRIMARY= 28	*Fault Tolerant CORBA* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01

Tag Name	**Tag Value**	**Described in**
ComponentId	TAG_ FT_HEARTBEAT_ENABLED = 29	*Fault Tolerant CORBA* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_MESSAGE_ROUTERS = 30	*CORBA Messaging* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_OTS_POLICY = 31	Object Transaction Service specification http://www.omg.org/technology/documents/formal/transaction_service.htm
ComponentId	TAG_INV_POLICY = 32	Object Transaction Service specification http://www.omg.org/technology/documents/formal/transaction_service.htm
ComponentId	TAG_CSI_SEC_MECH_LIST = 33	*Secure Interoperability* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_NULL_TAG = 34	*Secure Interoperability* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_SECIOP_SEC_TRANS = 35	*Secure Interoperability* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_TLS_SEC_TRANS = 36	*Secure Interoperability* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_ACTIVITY_POLICY = 37	Additional Structuring Mechanisms for the OTS: http://www.omg.org/technology/documents/formal/add_struct.htm
ComponentId	TAG_RMI_CUSTOM_MAX_STREAM_FORMAT = 38	Java to IDL Language Mapping specification: http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm
ComponentId	TAG_GROUP = 39	Unreliable Multicast (orbos/01-03-01)
ComponentId	TAG_GROUP_IIOP = 40	Unreliable Multicast (orbos/01-03-01)
ComponentId	TAG_PASSTHRU_TRANS = 41	Firewall Traversal specification (ptc/04-03-01)
ComponentId	TAG_FIREWALL_PATH = 42	Firewall Traversal specification (ptc/04-03-01)
ComponentId	TAG_IIOP_SEC_TRANS = 43	Firewall Traversal specification (ptc/04-03-01)
ComponentId	TAG_HOME_LOCATION_INFO = 44	Telecom Wireless specification: see http://www.omg.org/technology/documents/formal/telecom_wireless.htm.
ComponentId	TAG_DCE_STRING_BINDING = 100	*The DCE ESIOP* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_DCE_BINDING_NAME = 101	*The DCE ESIOP* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01
ComponentId	TAG_DCE_NO_PIPES = 102	*The DCE ESIOP* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01

Tag Name	Tag Value	Described in
ComponentId	TAG_DCE_SEC_MECH = 103	Security Service specification: see http://www.omg.org/technology/documents/formal/security_service.htm
ComponentId	TAG_INET_SEC_TRANS = 123	Security Service specification: see http://www.omg.org/technology/documents/formal/security_service.htm

A.4 Policy Type Tags

The table below lists the standard policy types that are defined by various parts of CORBA and CORBA Services in this version of CORBA/IIOP.

Policy Type	Policy Interface	Defined in	Uses create _policy
SecClientInvocationAccess = 1	SecurityAdmin::AccessPolicy	Security Service specification: see http://www.omg.org/technology/documents/formal/security_service.htm	N
SecTargetInvocationAccess = 2	SecurityAdmin::AccessPolicy		N
SecApplicationAccess = 3	SecurityAdmin::AccessPolicy		N
SecClientInvocationAudit = 4	SecurityAdmin::AuditPolicy		N
SecTargetInvocationAudit = 5	SecurityAdmin::AuditPolicy		N
SecApplicationAudit = 6	SecurityAdmin::AuditPolicy		N
SecDelegation = 7	SecurityAdmin::Delegation Policy		N
SecClientSecureInvocation = 8	SecurityAdmin::SecureInvocation Policy		N
SecTargetSecureInvocation = 9	SecurityAdmin::SecureInvocation Policy		N
SecNonRepudiation = 10	NRService::NRPolicy		N
SecConstruction = 11	CORBA::SecConstruction	*ORB Interface* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01	N
SecMechanismPolicy = 12	SecurityLevel2::MechanismPolicy	Security Service specification: see http://www.omg.org/technology/documents/formal/security_service.htm	Y
SecInvocationCredentialsPolicy = 13	SecurityLevel2::InvocationCredentials Policy		Y
SecFeaturesPolicy = 14	SecurityLevel2::FeaturesPolicy		Y
SecQOPPolicy = 15	SecurityLevel2::QOPPolicy		Y

Policy Type	**Policy Interface**	**Defined in**	**Uses create _policy**
THREAD_POLICY_ID = 16	PortableServer::ThreadPolicy	*Object Adapter* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01	Y
LIFESPAN_POLICY_ID = 17	PortableServer::LifespanPolicy		Y
ID_UNIQUENESS_POLICY_ID = 18	PortableServer::IdUniquenessPolicy		Y
ID_ASSIGNMENT_POLICY_ID = 19	PortableServer::IdAssignmentPolicy		Y
IMPLICIT_ACTIVATION_POLICY_ID = 20	PortableServer::ImplicitActivation Policy		Y
SERVENT_RETENTION_POLICY_ID = 21	PortableServer::ServentRetention Policy		Y
REQUEST_PROCESSING_POLICY_ID = 22	PortableServer::RequestProcessing Policy		Y
REBIND_POLICY_TYPE = 23	Messaging::RebindPolicy	*Asynchronous Messaging* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01	Y
SYNC_SCOPE_POLICY_TYPE = 24	Messaging::SyncScopePolicy		Y
REQUEST_PRIORITY_POLICY_TYPE = 25	Messaging::RequestPriorityPolicy		Y
REPLY_PRIORITY_POLICY_TYPE = 26	Messaging::ReplyPriorityPolicy		Y
REQUEST_START_TIME_POLICY_TYPE = 27	Messaging::RequestStartTimePolicy		Y
REQUEST_END_TIME_POLICY_TYPE = 28	Messaging::RequestEndTimePolicy		Y
REPLY_START_TIME_POLICY_TYPE = 29	Messaging::ReplyStartTimePolicy		Y
REPLY_END_TIME_POLICY_TYPE = 30	Messaging::ReplyEndTimePolicy		Y
RELATIVE_REQ_TIMEOUT_POLICY_TYPE = 31	Messaging::RelativeRequestTimeout Policy		Y
RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32	Messaging::RelativeRoundtripTimeout Policy		Y
ROUTING_POLICY_TYPE = 33	Messaging::RoutingPolicy		Y
MAX_HOPS_POLICY_TYPE =34	Messaging::MaxHopsPolicy		Y
QUEUE_ORDER_POLICY_TYPE = 35	Messaging::QueueOrderPolicy		Y
FIREWALL_POLICY_TYPE = 36	Firewall::FirewallPolicy	Firewall Traversal specification (ptc/04-03-01)	Y
BIDIRECTIONAL_POLICY_TYPE = 37	BiDirPolicy::BidirectionalPolicy	*General Inter-ORB Protocol* chapter: see CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01	Y

Policy Type	Policy Interface	Defined in	Uses create _policy
SecDelegationDirectivePolicy = 38	SecurityLevel2::DelegtionDirective Policy	Security Service specification: see http://www.omg.org/technology/documents/formal/security_service.htm	Y
SecEstablishTrustPolicy = 39	SecurityLevel2::EstablishTrustPolicy		Y
PRIORITY_MODEL_POLICY_TYPE = 40	RTCORBA::PriorityModelPolicy	Real-Time CORBA specification: http://www.omg.org/technology/documents/formal/real-time_CORBA.htm	Y
THREADPOOL_POLICY_TYPE = 41	RTCORBA::ThreadpoolPolicy		Y
SERVER_PROTOCOL_POLICY_TYPE = 42	RTCORBA::ServerProtocolPolicy		Y
CLIENT_PROTOCOL_POLICY_TYPE = 43	RTCORBA::ClientProtocolPolicy		Y
PRIVATE_CONNECTION_POLICY_TYPE = 44	RTCORBA::PrivateConnectionpolicy		Y
PRIORITY_BANDED_CONNECTION_POLICY_TYPE = 45	RTCORBA::PriorityBanded ConnectionPolicy		Y
TransactionPolicyType = 46	CosTransactions::TransactionPolicy	Object Transaction Service specification: http://www.omg.org/technology/documents/formal/transaction_service.htm	Y
REQUEST_DURATION_POLICY_TYPE = 47		*Fault Tolerant CORBA*: CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01.	
HEARTBEAT_POLICY_TYPE = 48		*Fault Tolerant CORBA*: CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01.	
HEARTBEAT_ENABLED_POLICY_TYPE = 49			
IMMEDIATE_SUSPEND_POLICY_TYPE = 50	valuetype MessageRouting:: ImmediateSuspend	*Asynchronous Messaging* chapter: CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01	N
UNLIMITED_PING_POLICY_TYPE = 51	valuetype MessageRouting::UnlimitedPing		N
LIMITED_PING_POLICY_TYPE = 52	valuetype MessageRouting::LimitedPing		N
DECAY_POLICY_TYPE = 53	valuetype MessageRouting::DecayPolicy		N
RESUME_POLICY_TYPE = 54	valuetype MessageRouting::ResumePolicy		N

Policy Type	**Policy Interface**	**Defined in**	**Uses create _policy**
INVOCATION_POLICY_TYPE = 55	CosTransactions::InvocationPolicy	Object Transaction Service specification: http://www.omg.org/technology/documents/formal/transaction_service.htm	Y
OTS_POLICY_TYPE = 56	CosTransactions::OTSPolicy		Y
NON_TX_TARGET_POLICY_TYPE = 57	CosTransactions::NonTxTargetPolicy		Y
ActivityPolicyType = 58	CORBA::PolicyType	Additional Structuring Mechanisms for the OTS: http://www.omg.org/technology/documents/formal/add_struct.htm	Y
OSA_MANAGER_POLICY = 59		Security Domain Membership (orbos/01-06-01)	
ODM_MANAGER_POLICY = 60			
PATH_SELECTION_POLICY_TYPE = 61		Firewall Traversal specification (ptc/04-03-01)	
PATH_INSERTION_POLICY_TYPE = 62			
PROCESSING_MODE_POLICY_TYPE = 63		*Portable Interceptor* chapter: CORBA, v3.0.3: http://www.omg.org/cgi-bin/doc?formal/04-03-01	

A.5 Exception Codes

If an exception that is to be raised for an error condition does not explicitly specify a specific standard minor code for that error condition, implementations can either use a minor code of zero, or use a vendor-specific minor code to convey more detail about the error.

The following table specifies standard minor exception codes that have been assigned for the standard system exceptions. The actual value that is to be found in the **minor** field of the **ex_body** structure is obtained by or-ing the values in this table with the **OMGVMCID** constant. For example “Missing local value implementation” for the exception NO_IMPLEMENT would be denoted by the **minor** value **0x4f4d0001**.

.

SYSTEM EXCEPTION	**MINOR CODE**	**EXPLANATION**
ACTIVITY_COMPLETED	1	Activity context completed through timeout or in some way other than requested
ACTIVITY_REQUIRED	1	Calling thread lacks required activity context

SYSTEM EXCEPTION	**MINOR CODE**	**EXPLANATION**
BAD_CONTEXT	1	IDL context not found
	2	No matching IDL context property
BAD_INV_ORDER	1	Dependency exists in IFR preventing destruction of this object
	2	Attempt to destroy indestructible objects in IFR
	3	Operation would deadlock
	4	ORB has shutdown
	5	Attempt to invoke ***send*** or ***invoke*** operation of the same ***Request*** object more than once
	6	Attempt to set a servant manager after one has already been set
	7	***ServerRequest::arguments*** called more than once or after a call to ***ServerRequest:: set_exception***
	8	***ServerRequest::ctx*** called more than once or before ***ServerRequest::arguments*** or after ***ServerRequest::ctx*** ***ServerRequest::set_result*** or ***ServerRequest::set_exception***
	9	***ServerRequest::set_result*** called more than once or before ***ServerRequest::arguments*** or after ***ServerRequest::set_result*** or ***ServerRequest::set_exception***
	10	Attempt to send a DII request after it was sent previously
	11	Attempt to poll a DII request or to retrieve its result before the request was sent
	12	Attempt to poll a DII request or to retrieve its result after the result was retrieved previously
	13	Attempt to poll a synchronous DII request or to retrieve results from a synchronous DII request
	14	Invalid portable interceptor call
	15	Service context add failed in portable interceptor because a service context with the given id already exists
	16	Registration of ***PolicyFactory*** failed because a factory already exists for the given ***PolicyType***

SYSTEM EXCEPTION	**MINOR CODE**	**EXPLANATION**
	17	***POA*** cannot create ***POA***s while undergoing destruction
	18	Attempt to reassign priority
	19	An OTS/XA integration **xa_start** call returned **XAER_OUTSIDE**
	20	An OTS/XA integration **xa_** call returned **XAER_PROTO**
	21	Transaction context of request and client threads do not match in interceptor
	22	Poller has not returned any response yet
	23	Registration of TaggedProfileFactory failed because a factory already exists for the given id
	24	Registration of TaggedComponentFactory failed because a factory already exists for the given id
	25	Iteration has no more elements
	26	Invocation of this operation not allowed in post_init
	27	Set_handle () has not been called previously on this listener instance
BAD_OPERATION	1	ServantManager returned wrong servant type
	2	Operation or attribute not known to target object

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
BAD_PARAM	1	Failure to register, unregister, or lookup value factory
	2	RID already defined in IFR
	3	Name already used in the context in IFR
	4	Target is not a valid container
	5	Name clash in inherited context
	6	Incorrect type for abstract interface
	7	string_to_object conversion failed due to bad scheme name
	8	string_to_object conversion failed due to bad address
	9	string_to_object conversion failed due to bad schema specific part
	10	string_to_object conversion failed due to non-specific reason
	11	Attempt to derive abstract interface from non-abstract base interface in the Interface Repository
	12	Attempt to let a ValueDef support more than one non-abstract interface in the Interface Repository
	13	Attempt to use an incomplete ***TypeCode*** as a parameter
	14	Invalid object id passed to ***POA::create_reference_by_id***
	15	Bad ***name*** argument in ***TypeCode*** operation
	16	Bad ***RepositoryId*** argument in ***TypeCode*** operation
	17	Invalid member name in ***TypeCode*** operation
	18	Duplicate label value in ***create_union_tc***
	19	Incompatible TypeCode of label and discriminator in ***create_union_tc***
	20	Supplied discriminator type illegitimate in ***create_union_tc***
	21	***Any*** passed to ***ServerRequest::set_exception*** does not contain an exception
	22	Unlisted user exception passed to ***ServerRequest::set_exception***
	23	wchar transmission code set not in service context
	24	Service context is not in OMG-defined range
	25	Enum value out of range
	26	Invalid service context Id in portable interceptor
	27	Attempt to call ***register_initial_reference*** with a null ***Object***
	28	Invalid component Id in portable interceptor
	29	Invalid profile Id in portable interceptor

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
	30	Two or more ***Policy*** objects with the same ***PolicyType*** value supplied to ***Object::set_policy_overrides*** or ***PolicyManager::set_policy_overrides***
	31	Attempt to define a ***oneway*** operation with non-void result, ***out*** or ***inout*** parameters or user exceptions
	32	DII asked to create request for an implicit operation
	33	An OTS/XA integration **xa_** call returned **XAER_INVAL**
	34	Union branch modifier called with bad case label discriminator
	35	Illegal IDL context property name
	36	Illegal IDL property search string
	37	Illegal IDL context name
	38	Non-empty IDL context
	39	Unsupported RMI/IDL custom value type stream format
	40	ORB output stream does not support ValueOutputStream interface
	41	ORB input stream does not support ValueInputStream interface
	42	Character support limited to ISO 8859-1 for this object reference
	43	Attempt to add a Pollable to a second PollableSet
BAD_TYPECODE	1	Attempt to marshal incomplete ***TypeCode***
	2	Member type code illegitimate in ***TypeCode*** operation
	3	Illegal parameter type
CODESET_INCOMPATIBLE	1	Codeset negotiation failed
	2	Codeset delivered in CodeSetContext is not supported by server as transmission codeset
DATA_CONVERSION	1	Character does not map to negotiated transmission code set
	2	Failure of **PriorityMapping** object
IMP_LIMIT	1	Unable to use any profile in IOR
INITIALIZE	1	Priority range too restricted for ORB
INTERNAL	1	An OTS/XA integration **xa_** call returned **XAER_RMERR**
	2	An OTS/XA integration **xa_** call returned **XAER_RMFAIL**
INTF_REPOS	1	Interface Repository not available
	2	No entry for requested interface in Interface Repository
INVALID_ACTIVITY	1	Transaction or Activity resumed in wrong context, or invocation incompatible with Activity’s current state
INV_OBJREF	1	wchar Code Set support not specified
	2	Codeset component required for type using ***wchar*** or ***wstring*** data

SYSTEM EXCEPTION	**MINOR CODE**	**EXPLANATION**
INV_POLICY	1	Unable to reconcile IOR specified policy with effective policy override
	2	Invalid ***PolicyType***
	3	No ***PolicyFactory*** has been registered for the given ***PolicyType***
MARSHAL	1	Unable to locate value factory
	2	***ServerRequest::set_result*** called before ***ServerRequest::ctx*** when the operation IDL contains a context clause
	3	***NVList*** passed to ***ServerRequest::arguments*** does not describe all parameters passed by client
	4	Attempt to marshal ***Local*** object
	5	***wchar*** or ***wstring*** data erroneosly sent by client over ***GIOP*** 1.0 connection
	6	***wchar*** or ***wstring*** data erroneously returned by server over ***GIOP*** 1.0 connection
	7	Unsupported RMI/IDL custom value type stream format
	8	Custom data not compatible with ValueHandler read operation
	9	Codeset service contexts with different values received on the same connection
	10	Custom data not compatible with ValueHandler 2.x read operation
NO_IMPLEMENT	1	Missing local value implementation
	2	Incompatible value implementation version
	3	Unable to use any profile in IOR
	4	Attempt to use DII on Local object
	5	Biomolecular Sequence Analysis iterator cannot be reset
	6	Biomolecular Sequence Analysis metadata is not available as XML
	7	Genomic Maps iterator cannot be rest
	8	Operation not implemented in local object
	9	Valuetypes not supported by CORBA-WSDL/SOAP implementation
	10	Valuetype sharing not supported by CORBA-WSDL/SOAP implementation
NO_RESOURCES	1	Portable Interceptor operation not supported in this binding
	2	No connection for request's priority
NO_RESPONSE	1	Reply is not available immediately in a non-blocking call

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
OBJ_ADAPTER	1	System exception in ***AdapterActivator::unknown_adapter***
	2	Incorrect servant type returned by servant manager
	3	No default servant available [POA policy]
	4	No servant manager available [POA Policy]
	5	Violation of POA policy by ***ServantActivator::incarnate***
	6	Exception in **PortableInterceptor::**IORInterceptor.components_established
	7	Null servant returned by servant manager
OBJECT_NOT_EXIST	1	Attempt to pass an unactivated (unregistered) value as an object reference
	2	Failed to create or locate Object Adapter
	3	Biomolecular Sequence Analysis Service is no longer available
	4	Object Adapter inactive
	5	This Poller has already delivered a reply to some client
TIMEOUT	1	Reply is not available in the Poller by the timeout set for it
	2	End time specified in RequestEndTimePolicy or RelativeRequestTimeoutPolicy has expired
	3	End time specified in ReplyEndTimePolicy or RelativeReplyTimeoutPolicy has expired
TRANSACTION_ROLLEDBACK	1	An OTS/XA integration **xa_** call returned **XAER_RB**
	2	An OTS/XA integration **xa_** call returned **XAER_NOTA**
	3	OTS/XA integration **end** was called with success set to **TRUE** while transaction rollback was deferred
	4	Deferred transaction rolled back
TRANSIENT	1	Request discarded because of resource exhaustion in ***POA***, or because POA is in discarding state
	2	No usable profile in IOR
	3	Request cancelled
	4	***POA*** destroyed
UNKNOWN	1	Unlisted user exception received by client
	2	Non-standard System Exception not supported
	3	An unknown user exception received by a portable interceptor

A.6 Identity Tokens

The following identity tokens must be powers of two.

- ITTAbsent = 0;
- ITTAnonymous = 1;

- ITTPrincipalName = 2;
- ITTX509CertChain = 4;
- ITTDistinguishedName = 8;
- ITTCompoundToken = 16;

INDEX