Spectra ORB C++ Edition User Guide

Version 2.1

PRISMTECH

Guide edition: 07 (07/11/16)

Table of Contents

1 Preface

1.1 About the User Guide

The *Spectra ORB (formally known as e*ORB) C++ Edition User Guide* provides instructions and information needed to use the product.

The *User Guide* should be read in conjunction with the *Product* and *IDL Guides*, as well as the other documents included with the product; please refer to the *Product Guide* for a complete list of documents.

1.2 Intended Audience

The *User Guide* is intended to be used by developers who use Spectra ORB to develop CORBA-based distributed applications.

1.3 Organisation

The *User Guide* is logically organized into two major parts. The first part of the guide describes the CORBA architecture including the Portable Object Adaptor (POA). In the second part of the guide the specifics of the Spectra ORB implementation are covered.

The first part of the guide is preceded by an *Introduction*, providing a high level description of the ORB's features. At the end of the guide Appendices are included providing details of each of the CORBA profiles supported by the product, there is also a *Bibliography,* listing a recommended set of useful resources.

The *CORBA* section of the guide is sub-divided into the following topics:

- Chapter *3, CORBA Basics*, provides a basic introduction to CORBA
- Chapter *4, Portable Object Adapter*, describes how to use the ORB's Portable Object Adapter

The section detailing product specifics is organised into these topics:

- Chapter *5, CORBA Based Extensions*, describes the additional CORBA features support by Spectra ORB which are not specified in the supported Profiles
- Chapter *6, Additional Features*, describes additional Spectra ORB proprietary features
- Chapter *7, Extensible Transport Framework*, covers Spectra ORB's Extensible Transport Framework (ETF)
- Chapter *8, Using the ORB*, provides information necessary to create applications which use Spectra ORB

1.4 Conventions

The conventions listed below are used to guide and assist the reader in understanding the Guide.

!	Item of special significance or where caution needs to be taken
i	Item contains helpful hint or special information
WIN	Information applies to Windows (*e.g.* XP, Vista, Windows 7) only
UNIX	Information applies to Unix based systems (*e.g.* Solaris) only
C	C language specific
C++	C++ language specific
Java	Java language specific

Hypertext links are shown as *blue italic underlined*.

On-Line (PDF) versions of this document: Items shown as cross-references to other parts of the document, *e.g.* *Contacts* on page 8, behave as hypertext links: jump to that section by clicking on the cross-reference.

```
% Commands or input which the user enters on the
command line of their computer  terminal
```

`Courier`, **`Courier Bold`**, or *`Courier Italic`* fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to "example" and
// kind field to an empty string

newName[0] = new NameComponent ("example", "");
rootContext.bind (newName, demoObject);
```

Italics and ***Italic Bold*** indicate new terms, or emphasise an item.

Sans-serif indicates user-related actions, *e.g.* File > Save (a sequence of selections from menus, or buttons or check-boxes).

Step 1: One of several steps required to complete a task.

1.5 Contacts

PrismTech can be contacted at the following contact points.

Corporate Headquarters	**European Head Office**
PrismTech Corporation 400 TradeCenter Suite 5900 Woburn, MA 01801 USA	PrismTech Limited PrismTech House 5th Avenue Business Park Gateshead NE11 0NG UK
Tel: +1 781 569 5819	Tel: +44 (0)191 497 9900
	Fax: +44 (0)191 497 9901

Web:	*http://www.prismtech.com*
Technical questions:	technical-support@*prismtech.com*
Sales enquiries:	*sales@prismtech.com*

2 Introduction

2.1.1 About Spectra ORB C++ Edition

Spectra ORB C++ Edition is a lightweight, high performance Object Request Broker (ORB) designed specifically for Distributed Real-time Embedded (DRE) systems and that can be configured to support different CORBA profiles based on a user's needs.

Spectra ORB C++ Edition v2 provides a pluggable set of libraries that can be used to configure the product to support the following CORBA profiles:

- Minimum CORBA Profile
- CORBA/e Compact Profile
- Software Communication Architecture v4.0 Full Profile

Specifically Spectra ORB provides support for CORBA APIs and features defined in the following standards:

- Minimum CORBA Specification version 1.0: OMG Document formal/02-08-01, August 2002
- Common Object Request Broker Architecture for Embedded (CORBA/e) Specification Version 1.0: OMG Document Number: formal/2008-11-06
- Software Communications Architecture Specification – Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS)– SCA v2.2.2 – FINAL / 15 May 2006
- Software Communications Architecture Specification V4.0, 28th February 2012, - Appendix E: Platform Specific Model (PSM) – Common Object Request Broker Architecture (CORBA)
- Interoperable Naming Service Specification: OMG Document formal/00-11-01
- Lightweight Log Service Specification: OMG Document formal/05-02-02: v1.1
- Event Service Specification: OMG Document formal/05-02-02: v1.1
- Real-time CORBA Specification, January 2005 Version 1.2 formal/05-01-04
- C++ Language Mapping Specification, July 2012, Version 1.3, formal/2012-07-02

Spectra ORB C++ Edition offers developers all of the advantages of standards-compliant, object oriented programming combined with high performance and a low footprint.

2.1.2 Key Features

- *Micro ORB kernel*
- *IDL to C++ compiler*
- *GIOP 1.2 support (with the exception of bi-directional support)*
- *Pluggable Portable Object Adaptor (POA) architecture* containing an extensible and highly scalable plug-in POA architecture (including child POAs)
- *Extensible Transport Framework* – providing multi-transport plug-in support, TCP transport by default; provides users with the ability to develop additional custom transports as required (e.g. UDP, RapidIO, Compact PCI, other..)
- *Native and portable exceptions*
- *Multi-thread safe*
- *User configurable server-side threading* – Thread-Per-Request and Thread-Per-Connection models supported
- *Pluggable Any data type support*
- *Request time-outs*
- *Pluggable Real-time CORBA support*
- *Suite of Lightweight Common Object Services* – Naming Service (both Full and Lightweight), Event Service, Log Service

2.1.3 CORBA Profiles

Spectra ORB was specifically designed for use in resource constrained embedded environments. In order to optimize memory footprint and performance it implements a number of specific subsets of functionality from the OMG's *CORBA Specification* required by DRE systems. These subsets of functionality are referred to as Profiles.

The OMG's *Minimum CORBA Specification* was the first CORBA Profile and was created in order to provide support for restricted environments where the full CORBA version was too large to meet the size and performance requirements. Domain specific standards such a the Software Communications Architecture (v2.2.2) mandate the use of ORB implementations that can support Minimum CORBA.

Minimum CORBA was eventually superseded by the OMG's *CORBA for Embedded Specification (CORBA/e)* which introduced two new Profiles; the CORBA/e Compact and Micro Profiles. The CORBA/*e* Compact Profile supports sophisticated applications such as real-time image and signal processing on board based systems running a standard Real-Time Operating System (RTOS). The CORBA/*e* Micro Profile supports basic functionality on the smallest networked systems, including Digital Signal Processors (DSPs) and the low-powered microprocessors found on typical hand-held devices.

The most recent version of *Software Communications Architecture Specification*, V4.0, defines a Platform Specific Model (PSM) specifying three new CORBA Profiles derived from CORBA/*e* with additional features from RT CORBA and a feature set tuned specifically for Software Defined Radio (SDR) applications.

The SCA CORBA profiles are characterized as follows:

1. SCA Full CORBA (Full) Profile – is the Full CORBA profile and is intended for SDR applications that will be hosted on most General Purpose Processor (GPP) platforms
2. SCA Lightweight CORBA (LW) Profile – is more constrained than the SCA Full CORBA Profile and is targeted towards environments with limited computing support such as a DSP.
3. SCA Ultra-Lightweight CORBA (ULW) Profile – is more constrained than the SCA Lightweight CORBA Profile and is specifically intended for processing elements with even more limited computing support such as those hosted on FPGAs.

2.1.3.1 Minimum CORBA Profile

The Minimum CORBA Specification, which refers to itself as the *minimumCORBA* specification, excludes features which were not felt to be essential in restrictive, high performance environments:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, recursive Valuetypes, dynamic Any – do not execute).
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any, recursive Valuetypes.
- Server-side: POA Supporting Transient or Persistent objects; Retained servants (disallows Implicit Activation).
- DCE ESIOP is omitted.
- Interworking between COM and CORBA is omitted.
- Interceptors are omitted.

All other features from the mandatory parts of OMG's CORBA Specification are supported in Minimum CORBA.

2.1.3.2 CORBA/e Compact Profile

Compact yet powerful: Fits resource-constrained systems (32-bit processor running a RTOS), but supports sophisticated applications such as signal or image processing in Real-time:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any – do not execute) with exception of Abstract Interface, Context clauses and Import.
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Supports Real-time CORBA with Static Scheduling; Propagates Real-time CORBA priorities over the wire.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Restricted Valuetypes – no Value Boxes or Custom Valuetypes.

- Messaging QoS - Rebind Support, Synchornisation Scope, Request and Reply Timeouts.
- Server-side: POA Supporting Transient or Persistent objects; Retained servants (disallows Implicit Activation); Prioritized multi-threading under ORB control.
- Includes Naming, Events, and Lightweight Logging Services.

2.1.3.3 CORBA/e Micro Profile

Truly Micro: Fits on a mobile or similar device with a lowpower microprocessor, or high-end DSP.

- Compiles all OMG IDL (Dynamic aspects of CORBA – IFR, DII, DSI, transient Servants – do not execute) with exception of Abstract Interface, Context clauses and Import.
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Messaging QoS - Rebind Policy, Synchronization Scope Policy, Request and Reply Timeout Policies
- Supports only statically defined Interfaces, Interactions, and Scheduling.
- Supports Real-time CORBA MUTEX interfaces.
- Server-side: For compactness and deterministic behavior supports exactly one POA; allows only transient, retained servants with unique, system-assigned IDs; and multithreading under ORB control.

2.1.3.4 SCA 4.0 Full Profile

Targets high performance SDR applications based on the SCA and typically hosted on a GPP:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, Valuetypes, dynamic Any – do not execute) with exception of WChar data type, Abstract Interface, Context clauses and Import.
- Restricted Any data type that can only contain:
  - Basic CORBA Types
  - Sequences of basic types (such as String)
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Supports Real-time CORBA with Static Scheduling; Propagates Real-time CORBA priorities over the wire.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Messaging QoS - Synchronization Scope Policy.
- Server-side: POA Supporting Transient or Persistent objects; Retained servants (disallows Implicit Activation); Prioritized multi-threading under ORB control.

- The Full Profile supports the additional standardized parameters identified in Table 1 to the ORB_init call to allow the root POA to be created with non-default policies.

Policy	**Default Value**	**Alternate Value**	**Optional Parameter to Override**	**Full Profile**	**LW Profile**
Lifespan Policy	TRANSIENT	PERSISTENT	-ORBPOAPersistent	✓	✓
ID Uniqueness Policy	UNIQUE_ID	MULTIPLE_ID	-ORBPOAMultipleId	✓	✓
ID Assignment Policy	SYSTEM_ID	USER_ID	-ORBPOAUserId	✓	✓

Table 1: ORB_init() Parameters

- Supports Real-time CORBA with Static Scheduling; Client Propagated and Server Declared Priority Models, RT Thread Pools.

2.1.3.5 SCA 4.0 Ligthtweight Profile

Targets high performance embedded SDR applications based on the SCA and hosted on an extremely resource limited processor such as a DSP:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, Valuetypes, dynamic Any – do not execute) with exception of Any and WChar data type, Abstract Interface, Context clauses and Import.
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Server-side: For compactness and deterministic behavior supports exactly one POA; allows only transient, retained servants with unique, system-assigned IDs; and multithreading under ORB control.
- The Lightweight Profile supports the additional standardized parameters identified in Table 1 to the ORB_init call to allow the root POA to be created with non-default policies.

For a detailed mapping between functionality defined in each CORBA profile and the functionality provided by Spectra ORB C++ Edition please refer to Table 4.

3 CORBA Basics

3.1.1 Introduction to CORBA

CORBA stands for Common Object Request Broker Architecture. CORBA is the Object Management Group's (OMG):

> *"open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network."* [1]

The Object Management Group is a non-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications.

3.1.2 The ORB

A core element of CORBA is the *Object Request Broker*, commonly referred to as the *ORB*.

An ORB mediates between an object and one of its clients. A client is defined as any computing context that invokes operations on the object (that is, sends it a message, or invokes a method). ORBs can take many different forms. In common practice, ORBs are mechanisms that mediate between clients and objects on different computers, using some kind of network communication. In this setting, ORBs are one of the principal enabling technologies in the field of distributed object computing.

3.1.2.1 Distributed Object Computing

Most popular object-oriented programming languages provide language constructs for encapsulation, inheritance, polymorphism, and other characteristic object-oriented concepts. These mechanisms have proven beneficial when building single-process applications. However, because they are implemented as programming language features, the benefits are not available when the application needs to interact with other processes or with remote machines. Programmers must generally resort to techniques such as sockets to build distributed applications.

Distributed object technology extends the benefits of object-oriented technology across process and machine boundaries to encompass entire networks. In short, this technology makes remote objects appear to programmers as if they were local objects (that is, simple programming-language objects in the same process). This effect can be described as location transparency.

1 The OMG's definition from its web site at *http://www.omg.org*.

3.1.2.2 Transparencies

Transparencies occur when a software abstraction allows programmers to cross a computing boundary (such as a boundary between different languages, machines, network protocols, and so on) without having to be aware of the boundary at all, or without performing an explicit transformation to cross it.

In an object system, location transparency means that an object's client can invoke the object's methods in a natural manner, regardless of where the object actually resides. The target object may reside in the client program itself (as is inherently the case with most object-oriented programming languages), it may reside in another address space on the same machine as the client, or it may reside on a remote machine. The object's programming interface (from the client's perspective) is identical in all cases. See *Figure 1* for an illustration of this concept.

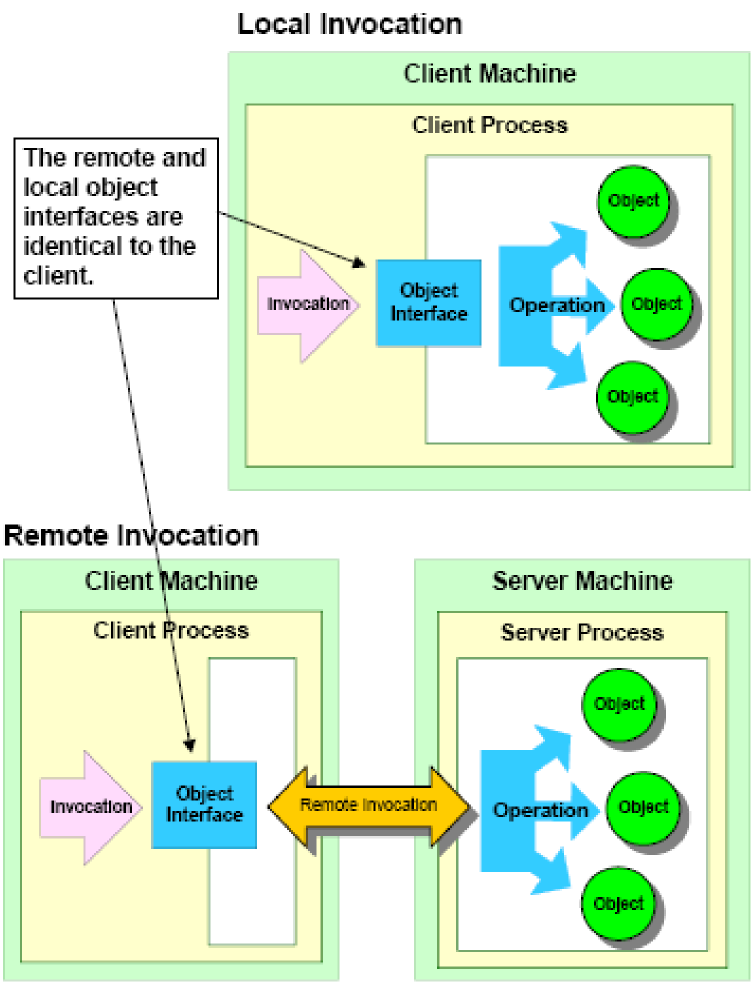


Figure 1 Remote Invocations and Location Transparency

In the CORBA model, the ORB provides the location transparency. ORBs also provide many other useful transparencies, including the following:

- **Programming language transparency**- The client and the object may be written in different programming languages and the ORB hides this fact; a Java client is completely unaware that it is invoking an operation on a language-specific object, whether Java, C++, or Smalltalk, and vice versa.

- **Platform transparency**- The client and object implementation programs may be executing on different types of computing hardware, with different operating systems, in such a way that both programs are unaware of these differences.
- **Representation transparency**- Because of language, hardware, or compiler differences, processes communicating through an ORB may have different low-level data representations. The ORB automatically converts different byte orders, word sizes, floating point representations, and so on, so that application programmers can ignore the differences and avoid problems.

As lower-level distribution problems become transparent, architects and programmers can focus their efforts on solving application problems, not plumbing problems. Expressed in other terms, distributed object technology raises the level of abstraction for distributed application design and development.

3.1.3 Distributed Object Computing and CORBA

OMG specifications have emerged as the primary focus of industry standardization in distributed object computing, client/server computing, and large-scale object-oriented application development. The CORBA specifications provide the foundation for the most comprehensive platform for system interoperability and software portability that is foreseeable in today's computing market.

To this end, CORBA specifies:

- a concrete object model
- an abstract language for describing object interfaces
- abstract programming interfaces for implementing, using, and managing objects
- equivalent concrete programming interfaces in popular object-oriented programming languages (that is, language mappings)
- operational interfaces between ORBs to ensure interoperability between products from different vendors

Other OMG specifications include CORBA services, which specifies standard interfaces for fundamental object services, such as naming and persistence, that are frequently required and generally useful for managing objects regardless of their function or application domain.

3.1.3.1 Interfaces

In the CORBA object model, attention is primarily focused on the object's interface.

An interface is the boundary layer that separates a consumer of an object's service (a client) from the supplier of the object's service (an object implementation). The interface defines what a client can know about an object and how a client may interact with it. As such, it hides the low-level details on one side of the boundary from the other side.

It may seem contradictory to describe interfaces as "hiding" things and providing "transparencies" at the same time, but it really isn't. The details that are hidden (such as network protocols, programming language idiosyncrasies, physical data organization, and so on) are like dirt on a window. They obscure what you really want to view—the abstract behaviour of the object. By wiping these details out of the way (or hiding them) ORBs give an object's consumer clear, un-obscured access to the object's essential behaviour, expressed in terminology natural to the consumer.

An interface may also be viewed as a *contract* between an object's client and implementation. The implementation agrees to respond to a given request with certain results; both the client and the implementation agree on the information that will be exchanged in a given operation, and so on. If both sides abide by the contract and don't rely on any assumptions that aren't stated explicitly in the contract, then the interaction between client and object will behave properly.

A CORBA interface consists of a collection of operations, attributes, and definitions for data types that are used with the operations and attributes. CORBA interfaces may be composed from other interfaces through inheritance.

Almost every section of the CORBA specification deals with one aspect of interfaces or another, such as how interfaces are described, how the descriptions are stored and managed, how abstract descriptions are mapped into concrete programming interfaces in various programming languages, how object implementations relate to and support an interface, and so on.

The CORBA specification defines a language for describing abstract object interfaces, called Interface Definition Language, or IDL.

3.1.3.2 Programming with CORBA Interfaces

IDL can be used to generate the stubs and skeletons that are actually used when programming. Since IDL is only an abstract interface description language, it must be transformed into equivalent constructs in a concrete programming language to be useful. The way in which these transformations are made for a particular language is called a *mapping* for that language.

Figure 2 illustrates the relationships between stubs, skeletons, clients, object implementations, and the ORB.

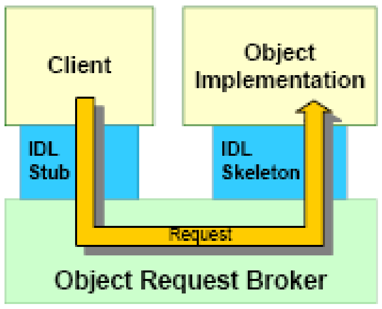


Figure 2 ORB Component Relationships

3.1.3.2.1 Stubs

Stubs are used by clients to invoke operations on target CORBA objects.

A stub is not the CORBA object itself. It *represents* a CORBA object and is, in part, responsible for propagating requests (invocations) made on itself to the real target object. In keeping with this role, stubs are sometimes called *proxies* or *surrogates*.

When the target object resides in a remote process, the stub is responsible for packaging the request, with its parameters, into a message to send to the remote process across a network, then receiving the reply message from the object, unpacking the operation results from the message, and returning them to the calling program.

3.1.3.2.2 Skeletons

Skeletons are used to build object *implementations*. An implementation of a CORBA interface is a package of code in a concrete programming language that provides the real behaviour of the object type. In some cases, the term implementation is used to indicate the body of code in an abstract sense, that is, the *type* (as opposed to an individual instance). In other cases, implementation can mean a specific instance of the implementation type. When there is a possibility of ambiguity, we will distinguish between the two as *implementation type* and *implementation instance*.

A skeleton takes the form of an abstract base class declaration with abstract functions that correspond to the operations in the IDL interface. Programmers construct an implementation by deriving a new type from the skeleton class, then providing method implementations for the operations inherited from the skeleton class.

The stub and skeleton have identical (or nearly identical) interfaces. They are type compatible (i.e., can be substituted for one another) at the level of the common base interface.

3.1.3.2.3 Clients and Servers

When a program includes the stub type and invokes operations on instances of the stub type, that program is acting in the role of a *client*, with respect to the target object represented by the particular stub instance. When a program includes an implementation type (derived from the skeleton), creates instances of the implementation type, and makes them available for use by clients, the program is acting in the role of *server*, with respect to the implemented objects.

Note that the terms client and server merely describe *roles* that programs play with respect to a particular object or set of objects. In a distributed object context (or more specifically, a CORBA context), these terms do not indicate architectural roles played by the programs, as they do in the traditional sense of client/server computing. A client of one CORBA object may be the server for other clients.

Programs sharing each others' objects in a variety of client/server roles may in fact be peers architecturally.

3.1.3.3 Delivering Requests Using and ORB

As described above, an ORB is anything that mediates between a client and its target object. By *mediate*, we mean to deliver the request from the client context to the server context, invoke the method on the target object, and deliver results, if any, back to the client. CORBA does not in any way prescribe or limit the mechanisms that an ORB may use to accomplish this task. The range of possible implementations is extremely large, and has interesting consequences, both practical and theoretical.

By leaving implementation decisions completely free, the CORBA specification allows highly specialized ORBs to be optimised for particular environments with unusual requirements, such as embedded real-time systems. For the purposes of this discussion, however, we will describe the Spectra ORB implementation.

3.1.3.3.1 Delivering Requests to Remote Objects

The ORB is a set of libraries that are linked into the client and server programs of the distributed CORBA-based application. When the client invokes an operation on the object, via the stub, the stub and the client-resident ORB library cooperate to assemble a message that describes the request. After assembling the message, the stub invokes the appropriate function in the client-resident class, transmitting the message to the server that contains the target object.

The message is received in the server by the server-resident ORB component. This component is responsible for decoding the message. The portable object adapter (POA) locates the specific object targeted in the request and passes the message contents to the skeleton. The skeleton extracts the request parameters and invokes the requested operation on the object implementation instance. The process then reverses itself: the skeleton creates the reply message, sends it back to the client, where the stub decodes it and returns the results to the client that made the request.

3.1.4 ORB Components

The ORB is composed of everything that intervenes between the client and the object to achieve location transparency. In a simple example, illustrated previously in *Figure 2*, the ORB encompasses the stub, the client-resident ORB classes, the server-resident ORB, and the skeleton. It can be argued that the network itself constitutes part of the ORB, because it mediates data transfer between processes - playing a major role in providing location transparency.

In an ORB's run-time environment, there may be a number of other processes (which are neither the client nor the server) that become involved in some aspect of the request delivery activity, to locate objects, start new server processes, monitor the status of requests in progress, and so on. It is usually not possible to point to a single process or software component and (accurately) call it *the ORB*.

Another way to determine what constitutes an ORB is to observe the two interface boundaries that the ORB mediates between. By boundary, we mean a specific API invocation (for example, function call, method invocation, and so on) through which non-ORB elements (clients and object implementations) interact with the ORB.

The client interacts with the ORB by invoking a member function on a stub. This boundary is labelled the client-ORB boundary in *Figure 3*. The object interacts with the ORB primarily by having one of its member functions invoked by the ORB.

This boundary is labelled the ORB-object boundary in the figure. Anything between those boundaries may be considered as part of the ORB for conceptual purposes.

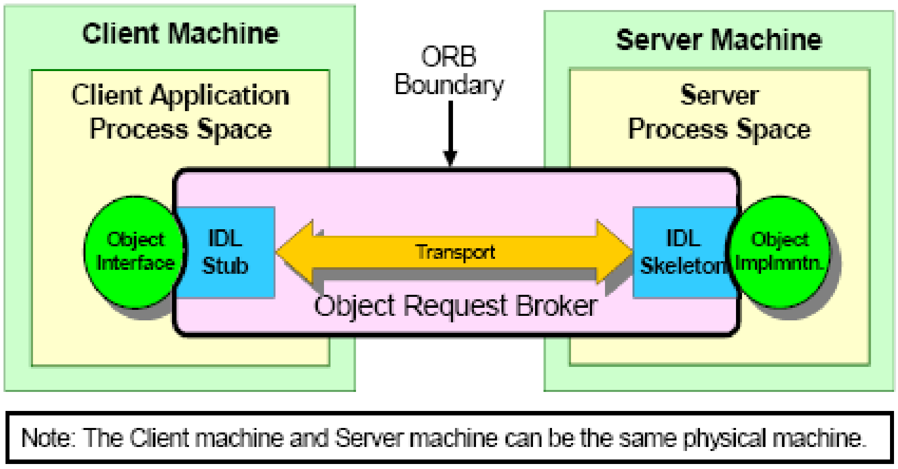


Figure 3 The ORB as an Abstraction

3.1.4.1 Abstraction

Contrast the previous example with the following scenario. As mentioned above, stubs and skeletons are derived from an interface. When a programmer uses an ORB-based object, methods are invoked on the common interface, not the derived stub or skeleton. Since both the stub class and the skeleton class (and, thus, the implementation class) are derived from the interface base class, client code that makes the invocation could be using either a stub that is bound to a remote object, or it could be invoking a method directly on an implementation instance that is in the same process. This use of C++ polymorphism allows the client to use remote and local objects in exactly the same way, without ever having to (or in some cases, even being able to) distinguish between them.

When a client "sends" a request to a local implementation instance, what constitutes the ORB? You might be tempted to say that there is no ORB present but, in fact, there is. All of the necessary elements are present - the client, the target object, and something that delivers the request from the client to the object. The delivery

mechanism (the ORB) in this case is the machine instruction that performs the function call on the target object's member function. The mediation between the client and the object takes place in a single stack frame in the local machine.

Thinking of this as an ORB may seem too abstract, but from the programmer's point of view a local invocation is indistinguishable (if the ORB is properly implemented) from a remote invocation. If it communicates like an ORB, it's an ORB.

If you consider this scenario with respect to interface boundaries, the client-ORB and ORB-object boundaries from the previous example have coalesced into a single client-ORB-object boundary, creating for us the mental image that the ORB (in the case of local invocations) is a two-dimensional, infinitely thin surface between the client and the server.

> *i* Spectra ORB always routes operation invocations via stubs. Local calls are not made directly on the implementation instance unless the idl is compiled with the "`collocated_direct`" compiler flag.

3.1.5 Terminology Explained

Figure 4 is an adaptation from the CORBA specification. The official OMG illustration of ORB architecture is modified to show only Minimum CORBA. The following subsections describe the elements shown in the figure and their roles in the overall activity of delivering requests. Some of the descriptions given here do not exactly match those in the CORBA specification. Where our descriptions vary, it is generally to achieve greater clarity and to provide a more consistent overall picture.

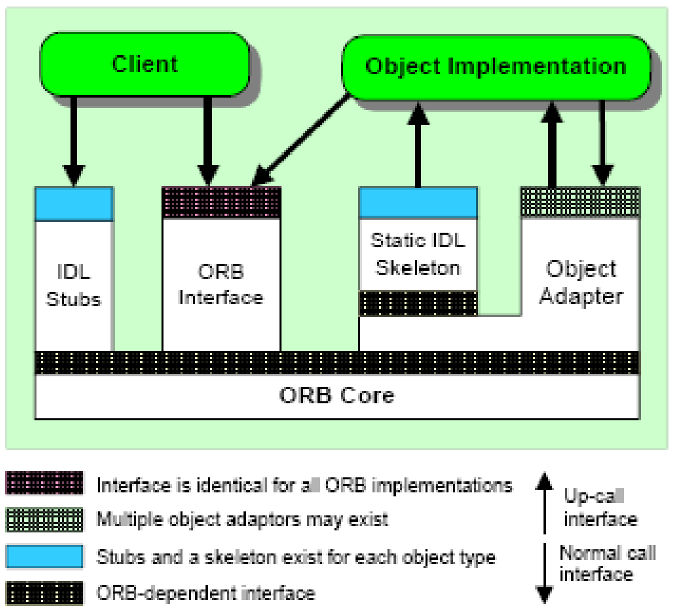


Figure 4 The Structure of Object Request Broker Interfaces

3.1.5.1 Clients and Servers

As mentioned above, the terms *client* and *server* in a distributed object context have a different meaning than the same terms used in the context of more traditional client-server computing. In CORBA, the terms refer primarily to roles played by different programs (or specific parts of programs) with respect to a particular object. The client of an object is the *processing context* from which a request is made on the object.

The term processing context is used advisedly, with some intentional ambiguity. Sometimes it may refer to the program (or process) that makes a request; it may also refer to a particular thread or a particular function from which an invocation is made. In some cases, it may refer to another object (an implementation instance) that contains a reference for the first object and makes requests on that object from within one of the containing object's methods. Though one object's methods may in fact constitute a client context for another object, there is formally no such thing as a *client object* in CORBA systems.

Likewise a *server* is the computing context in which an object is implemented. Sometimes the word server is used to indicate the object itself; other times it may denote the process in which an object resides. In general, its ambiguity is similar to that of the term *client*. Note again that the terms client and server apply to *roles* that components play, not the components themselves. Any given program may simultaneously be a client of some objects and a server for other (or the same) objects.

3.1.5.2 Object References

The meaning of the term *object reference* is relative to the context in which it is used. When used in a programming context in the ORB, an object reference takes the form of a C++ interface. Programmatic object references may also be converted into character strings, which may be later converted back into object references. These strings capture the information model encapsulated in the programmatic reference. Even though the string is not usable as a reference in a program, it is thought of as an object reference because it potentially locates and identifies a particular implementation instance.

The term object reference may be used to denote the abstract concept of an object's identity and location. In the process of handling requests, the ORB maintains internal data structures that it uses to locate, identify, and connect to the target objects. Since these structures are opaque to ORB users, they may be discussed only as an abstraction. One might say, for instance, that an object reference is passed from a client to a server as a parameter in an invocation. The thing being passed inside the ORB is neither the stub nor the reference in string form. Though you may not know its concrete form, it is sometimes useful to refer to this abstraction in discussions as an object reference.

3.1.5.3 First Class Objects and Pseudo Objects

In CORBA terminology, a first class object is a fully functional CORBA object supporting all of the attributes ascribed to regular CORBA objects:

- It has a unique identity assigned and managed by the ORB
- The ORB can supply references to the object that can be used by remote clients to make invocations on the object through the ORB
- It supports at least one CORBA interface described in IDL
- Its references support all of the operations defined on `CORBA::Object`
- It behaves in a manner consistent with general descriptions of objects in the CORBA specification

For various reasons, the CORBA specification and some CORBA services specifications define programming interfaces that, while object-oriented in style, cannot satisfy the requirements of a first-class object. In some cases the object is, of necessity, local to the process in which it is used; in other cases the interface cannot be properly expressed in IDL. In general, pseudo interfaces are used to provide APIs for ORB components or utility objects specific to ORB or service functions, such as the ORB interface itself or the interface for the POA. Pseudo interfaces generally become programming objects in the language mappings (that is, a class in C++), but do not support required righteous object behaviours, such as:

- They cannot be remotely accessed
- They do not have real object references (although they do have programmatic references)
- They do not support `CORBA::Object` operations

Another characteristic of pseudo objects is that their interfaces are often described in pseudo-IDL, or PIDL. PIDL is not really a language at all; it is more of a dialect of IDL that is used to describe interfaces for pseudo objects in a convenient, familiar manner, while recognizing that the PIDL need never actually be compiled into stubs and skeletons. Because this is the case, some pseudo interfaces described in PIDL contain syntax or data types that are not legal IDL but are intended to describe interface elements that are not allowed for righteous objects (hence, the need for pseudo objects). The following subsections describe some of the more important pseudo-objects.

3.1.5.3.1 The ORB Pseudo Object

The definition of ORB - given above - described the ORB as an abstract functional entity that mediates requests. The CORBA specification also describes a programming interface called the *ORB pseudo object*. This interface supports operations that interact with the computing environment provided by the CORBA implementation (the ORB in the abstract sense) such as initialization, and operations that perform utility functions, such as converting object references to and from strings. Although this pseudo object interface is called the ORB and it is a component of the abstract ORB entity, do not confuse the ORB pseudo object with the actual ORB, or infer from the way the interface is described that the ORB is a physical, identifiable object.

3.1.5.3.2 Object Adapters

The CORBA specification describes pseudo objects called *object adapters* that provide part of the interface between the ORB and object implementations. In particular, CORBA specifies an interface for the POA. The POA interface supports the following capabilities:

- It allows implementations to associate ORB-managed object identities with instances of user-supplied implementation classes
- It allows an implementation to inform the ORB that it (or one of its instances) has undergone a state change that affects its relationship with the ORB, such as activation (that is, the implementation or object is prepared to receive requests) or deactivation (the object is not available to receive requests)

4 Portable Object Adapter

The Portable Object Adapter is the link between the ORB and individual servants created in various programming languages. It is responsible for creating object references and for routing requests from the ORB to the appropriate servant.

The CORBA specification defines the Portable Object Adapter (POA) with the following features:

- source-level portability between ORB products
- allows multiple and distinct instances of the POA to exist in a server
- allows individual servants to support multiple object identities simultaneously
- provides a mechanism by which policy information can be associated with individual POA instances
- supports both persistent and transient objects

All references to the POA in this section describe the POA characteristics supported by Spectra ORB with the exception of the object adapter's dynamic behaviour which is not supported.

4.1 How The POA Works

In simplistic terms, after the client obtains an object reference it invokes a request on that object. That request is transmitted via the ORB to the server application. Refer to Figure 5, *Request Dispatching*. The POA is responsible for routing the request to the appropriate servant, which incarnates the target object responsible for processing the request.

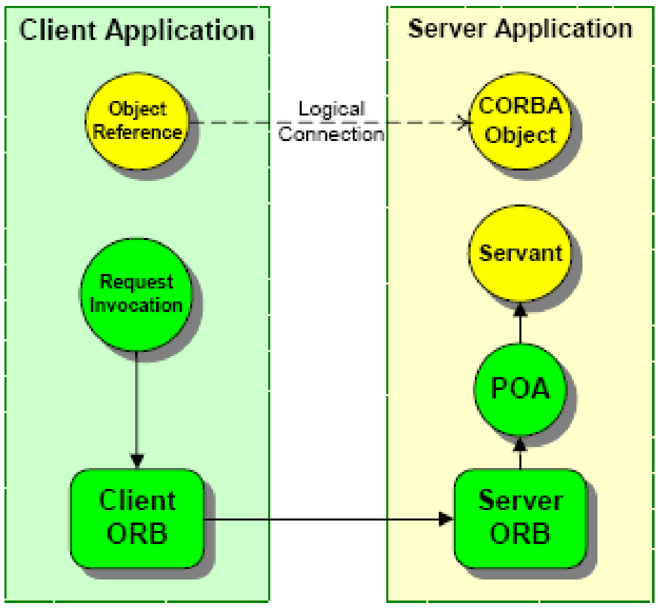


Figure 5 Request Dispatching

The POA maintains an association between the `ObjectId` (embedded in the object reference) and the servant (a programming language implementation of a CORBA object). This association is maintained in a table called the Active Object Map.

When a request is received, the object adapter looks at the `ObjectId` that came with the request and finds the servant associated with that `ObjectId` from its Active Object Map. Then it dispatches the request on that servant. A CORBA server process can contain a number of different POAs, each having their own Active Object Map. POAs are created in a hierarchical fashion, with the special `RootPOA` serving as a common ancestor to all other POAs.

The ability to create multiple POAs and to set characteristics on the POA using policies allows you to control POA behaviour and, consequently, the scalability and performance of your application.

4.2 POA Policies

Key to the POA definition is the ability to create multiple POAs and to customize each instance by setting policies. In general, you will define a list of policies, then assign them to a POA when it is created. Once a POA is created with an assigned set of policies, those policies cannot be changed for the life of the POA. A new POA does not inherit policies from its parent POA.

Interfaces that define policies to be assigned to a POA must be derived from `CORBA::Policy`.

4.2.1 Standard POA Policies

The following standard POA policies are supported by Spectra ORB.

4.2.1.1 Lifespan Policy

`POA::create_lifespan_policy` allows you to specify the lifespan of objects.

- `TRANSIENT` objects cannot outlive the processes in which they are first created.
- `PERSISTENT` objects can outlive the process in which they are created.

The default value for this policy is TRANSIENT.

Setting the TRANSIENT policy does not prevent explicit reactivation of a servant with the same object key. Change the object keys to enforce transient behaviour. The easiest way to do this is to create new POAs for servant reactivation.

4.2.1.2 Object Id Uniqueness Policy

POA::create_id_uniqueness_policy specifies whether servants activated by the POA must have unique ObjectIds.

- UNIQUE_ID specifies that each servant activated by that POA can support only one ObjectId.
- MULTIPLE_ID specifies that servants activated by that POA can support more than one ObjectId.

The default value for this policy is UNIQUE_ID.

4.2.1.3 Id Assignment Policy

POA::create_id_assignment_policy specifies whether ID assignment is performed by the POA or by the application.

- SYSTEM_ID specifies that the POA generates and assigns Object Ids.
- USER_ID specifies that ObjectIds are assigned by the application.

The default value for this policy is SYSTEM_ID.

4.2.1.4 ServerProtocol Policy

The RTCORBA::ServerProtocolPolicy is used to select and configure one or more communication protocols used by a POA. An instance of this policy object is created using the create_server_protocol_policy() operation on the RTCORBA::RTORB interface. The RTCORBA::RTORB interface can be obtained using ORB::resolve_initial_references("RTORB"). An alternative method for selecting and configuring communication protocols is by using the -ORBListenEndpoints command line option. As this policy effects core ORB functionality it is supported in the core ORB implementation and can be allocated directly with new.

i It is not necessary to add the RTORB plugin or link with the e_rtorb library in order to use the ServerProtocolPolicy.

4.2.2 POA Policy Summary

All POA policy objects are locality-constrained; that is, you cannot pass their references as arguments to normal CORBA operations or convert them to strings using ORB::object_to_string. They can be accessed only within the context of the ORB in which they were created.

Once you define the policies to be assigned to a POA, you can create the POA by calling create_POA on an existing POA. The new POA becomes the child of the POA on which the call was made. create_POA takes three arguments: the name for the new POA, a reference to the POAManager for that POA, and a list of policies to be applied to the new POA. If no POAManager is specified, a new POAManager is created.

4.3 POA Manager

The POAManager controls the flow of requests to one or more POA objects. The `POAManager` interface supports operations to change the state of a POA to one of the following:

POA	State Meaning
`ACTIVE`	Calling `activate` on the POAManager allows requests to flow to the POAs that it controls.

Table 2: POA States

i The POAManager interface also specifies `HOLDING`, `DISCARDING`, and `INACTIVE` states but these states are not supported by Spectra ORB. Only the `activate` operation and the `adapterInactive` exception are supported.

4.4 Object References, Keys, and Ids

The POA is responsible for creating an object reference, which the client can use to contact the target object. The object key is embedded within the object reference and the object identifier is embedded within the object key. The policies you set on the POA determine whether or not your application controls the content of the `ObjectId` and whether servants can support multiple IDs. `ObjectIds` must be unique within each individual POA; however different POAs can assign the same `ObjectId`.

4.5 Servants

The IDL compiler generates server-side skeleton classes. These skeletons are abstract base classes from which your servant classes are derived. Servant classes are obliged to implement all of the pure virtual functions declared in the generated skeletons. Servants are responsible for incarnating CORBA objects. A servant is a C++ instance used to service a request.

4.6 Object Creation and Activation

A CORBA object must be created and activated before the client can invoke operations on it. The POA remembers the relationship between the object and the servant which created it.

Depending on the policies set on the POA, you will either:

- use `POA::activate_object` or `POA::activate_object_with_id` to activate the object. Once the object is activated, the POA can dispatch requests arriving for that object. After activation, you may use the `_this()` or `POA::servant_to_reference()` operation to obtain an object reference from the servant.

or

- use `POA::create_reference_with_id` to create an object reference without activating it

However, because Spectra ORB does not support implicit activation of objects, you must explicitly activate the object to process requests to it. Use deactivate object to remove the association of the object with its servant.

4.7 Request Processing

When the ORB receives a request, it attempts to locate the appropriate POA and deliver the request. It uses the received object reference, which contains the `ObjectId` and POA identification, to locate the appropriate server and POA within that server. The request is then handed off to the POA.

The POA now takes over and tries to locate the target object. The POA searches for the servant associated with the `ObjectId` in its Active Object Map. Once a reference to the servant is obtained, the appropriate method is invoked. Otherwise, an exception is thrown.

4.8 Designing an Application

Spectra ORB does not support the dynamic features supported by the full-version of the CORBA specification. As a result, the minimum CORBA POA does not support certain features, such as *activation on demand*, which must be taken into account when designing servers. For example, POAs are not activated on demand because Spectra ORB does not support adapter activators (a POA must be activated when a server is started). The server must also be designed to activate all objects on which clients can invoke requests. The process of activation registers a servant associated with an `ObjectId` in the Active Object Map. If a request arrives for a servant that is not active, the POA will throw an exception.

5 CORBA Based Extensions

This section describes features which are part of the full CORBA specification and are provided by Spectra ORB, but which are not specified in the supported Profiles.

5.1 URL Object References

There are several standard portable ways for applications to obtain object references.

- A small set of initial references to essential objects, `RootPOA` and `POACurrent`, are available to applications using the ORB reference resolution operation, `resolve_initial_references().`
- If a Naming Service is available, it can be resolved as an initial reference then additional object references can be obtained from it.
- Alternatively a server may use the ORB `object_to_string()` method to convert an object reference to a string and then make the string available to clients by displaying it or writing it to a file. Clients can convert the string to an object reference by calling the ORB `string_to_object()` method.
- The URL object reference provides an additional way for clients to obtain object references.

A Uniform Resource Locator (URL) is a networked extension of the standard directory/filename. Object references can be constructed from a `corbaloc` type URL:

```
corbaloc:[<protocol>]:<endpoint>/<key>
```

where:

`<protocol>` is the protocol used, for example, `iiop`. Note that `iiop` is a special case in that it is the default protocol so need not be set.

`<endpoint>` is the endpoint for the specified protocol.

<key> is the object key (or short key). A short key is the name by which the object has been registered with the ORB with the `register_initial_reference` operation.

For the iiop protocol the endpoint has the format:

```
<host>:[<port>]
```

`<host>` is the internet host IP address or DNS-style name of the target server.

`<port>` is the port number where the object receives connections. The default value, `2908`, is used when `<port>` is omitted.

For example: `corbaloc:iiop:10.1.0.28:1234/Myserver`

5.1.1 Client Side Implementation

To use a URL to locate an object, a client must do the following:

Step 1: Start the application with a command line that includes the `-ORBInitRef` option in the form:

```
-ORBInitRef <name>=<url>
```

Step 2: Separate sequences of `-ORBInitRef <name>=<url>` are required for each object that the client will locate by URL. For example:

```
% Client -ORBInitRef payroll_server=corbaloc:iiop:money:8872/Pyroll \
         -ORBInitRef document_server=corbaloc:iiop:docs:8213/DocSvr
```

Step 3: Call `CORBA::ORB_init(argc, argv)`. The ORB converts the URLs to object references and adds them to the list of initial references.

Step 4: Call `CORBA::ORB::resolve_initial_references(<name>)` to obtain a reference to the object. `resolve_initial_references()` returns a generic `CORBA::Object_ptr` that must be narrowed to the object's actual type.

5.1.2 Server Side Implementation

Objects created by a server are not automatically accessible by URL. In order to make an object accessible via URL, a server must:

Step 1: Create and activate a servant in a POA.

Step 2: Get a reference to the servant with the POA `servant_to_reference` operation.

Step 3: Register the reference with the ORB `register_initial_reference` operation. The name by which the reference is registered can then be used as the key to the corbaloc URL by the client.

5.2 Tie Classes

A tie class is a servant that relays method calls. Specifically, all calls to an IDL interface's methods are relayed to a class of your choosing. The IDL compiler generates the tie class; you write a small amount of code, which tells the tie class to relay calls to your legacy class.

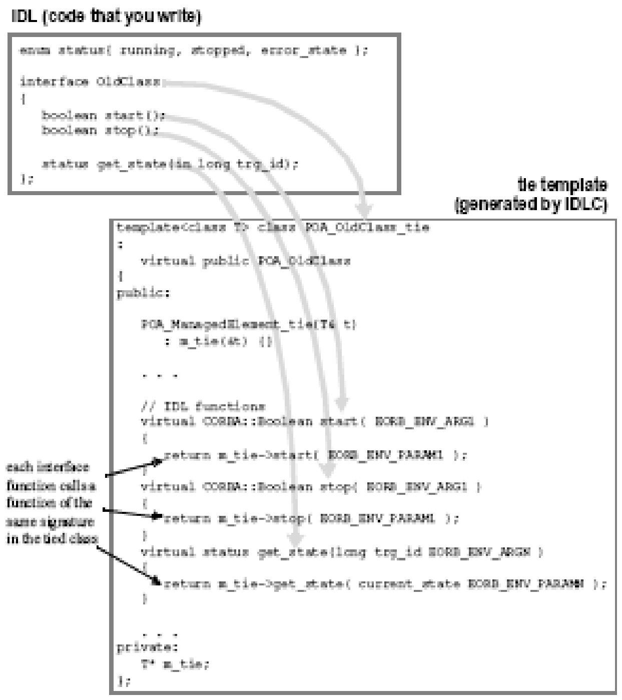
Connecting to the ORB via a tie class enables you to implement an IDL interface in a class that does not inherit from the ORB's class hierarchy. This is useful:

- To make legacy code accessible through CORBA when your legacy class is part of a large inheritance hierarchy. If a legacy class that you want to make accessible through CORBA is part of a large inheritance hierarchy, you might not want to add the burden of inheriting ORB classes. In addition, a tie class lets you call the legacy class through CORBA without modifying the legacy class.
- If storing objects in an OODB. In an object-oriented environment, it is not advisable to inherit the data and classes from the ORB, as it would be a needless addition to the size of the database.

5.2.1 How Tie Classes Work

As shown in Figure 6, *Generation of Tie Classes*, for each interface in your IDL, the IDL compiler generates a tie class that contains a function for each method in the interface. Each interface function simply calls a function of the same signature in the tied class (the class that you specify).

Figure 6 Generation of Tie Classes


IDL (code that you write)
tie template (generated by IDLC)
each interface function calls a function of the same signature in the tied class

```
enum status{ running, stopped, error_state };

interface OldClass
{
   boolean start();
   boolean stop();

   status get_state(in long trg_id);
};
```

```
template<class T> class POA_OldClass_tie
:
   virtual public POA_OldClass
{
public:

   POA_ManagedElement_tie(T& t)
      : m_tie(&t) {}

   . . .

   // IDL functions
   virtual CORBA::Boolean start( EORB_ENV_ARG1 )
   {
      return m_tie->start( EORB_ENV_PARAM1 );
   }
   virtual CORBA::Boolean stop( EORB_ENV_ARG1 )
   {
      return m_tie->stop( EORB_ENV_PARAM1 );
   }
   virtual status get_state(long trg_id EORB_ENV_ARGN )
   {
      return m_tie->get_state( current_state EORB_ENV_PARAMN );
   }

   . . .
private:
   T* m_tie;
};
```

Rather than generating a tie class directly, the IDL compiler creates a template class that you instantiate to define the tie class. For example, if you had a file `example.idl` containing an interface called `ExampleInterface`, then the IDL compiler would generate the following template definition in `example_s.h`:

To create an instance of the tie class, instantiate the template as follows:

```
POA_ExampleInterface_tie<ExampleClass> tieobj(*exampleobj, EORB_ENV_VARN);
```

This creates an object `tieobj` of a tie class that relays all `ExampleInterface` methods to `ExampleClass`.

By default the IDL compiler does not generate tie templates. To generate tie templates, run the IDL compiler using the `-gen_tie` flag. For example:

```
% idlcpp -gen_tie example.id
```

generates a tie template class for each interface in `example.idl`.

5.2.2 Tying a Legacy Class into CORBA

To make an existing class `OldClass` accessible through CORBA:

Step 1: Write an IDL interface definition for `OldClass`, giving the interface methods the same names as `OldClass'`s existing public methods.

Step 2: In your server code, wherever you create an `OldClass` object, tie it to CORBA by creating a tie object and activating the tie object like this:

```
OldClass* oldobj = new OldClass;
POA_OldClass_tie<OldClass> tieobj (*oldobj);
CORBA::ObjectID oid = rootPOA->activate_object (& tieobj);
```

The POA knows only about the tie class, `POA_OldClass_tie<OldClass>`, not about `OldClass`. `POA_OldClass_tie<OldClass>` delegates calls to `OldClass`.

5.2.3 Methods Supported

In addition to the IDL interface methods, a tie class supports the following functions:

```
POA_A_tie()
POA_A_tie(T& t)
POA_A_tie(T& t, PortableServer::POA_ptr poa)
POA_A_tie(T* tp, CORBA::Boolean release = 1)
POA_A_tie(T* tp, PortableServer::POA_ptr poa, CORBA::Boolean release = 1)
```

Constructors: If you pass the tied object directly, the constructor stores a pointer to it. If you pass a pointer to the tied object, the release flag tells whether the tie class should delete the pointer in the destructor.

```
T* _tied_object()
```

The `_tied_object()` accessor function allows callers to access the existing classes (C++ object). If the tie was constructed to take ownership of the C++ object (release was `TRUE` in the `T*` constructor), the caller of `_tied_object()` should never delete the return value.

```
void _tied_object(T& obj)
void _tied_object(T* obj, CORBA::Boolean release = 1)
```

The first `_tied_object()` modifier function calls delete on the current tied object if the tie's release flag is `TRUE`, and then points to the new tie object passed in. The tie's release flag is set to `FALSE`. The second `_tied_object()` modifier function does the same, except that the final state of the tie's release flag is determined by the value of the release argument.

```
CORBA::Boolean is_owner()
```

The `_is_owner()` accessor function returns `TRUE` if the tie owns the existing class (C++ object), or `FALSE` if it does not.

```
void _is_owner (CORBA::Boolean b)
```

The `_is_owner()` modifier function allows the state of the tie's release flag to be changed. This is useful for ensuring that memory leaks do not occur when transferring ownership of tied objects from one tie to another, or when changing the tied object.

5.2.4 Specification Compliance

The ORB's implementation of tie templates is based on the *C++ Language Mapping Specification*, June 1999, aligned with version 2.3 of the *CORBA Specification*, with the exception of method `POA_ptr _default_POA` which is not overridden by tie classes.

6 Additional Features

This section describes the additional, proprietary features provided by Spectra ORB

Features and topics covered include:

- the TCP transports provision
- the ORB's libraries
- proprietary ORB extensions
- object rebinding
- Portable Server policy definitions
- `_this()` method behaviour under Minimum CORBA
- exceptions, including

  - platforms which support C++ exceptions

  - platforms without C++ exception support

  - the Portable Exception Layer Macros which enables seamless cross-platform programming by providing C++-style exception handling on platforms which do not support C++ exceptions
- logging

6.1 Extensible Transport Framework

The ORB supports a pluggable transport layer that provides users with the ability to develop additional, custom transports as required.[1] Please note that a default TCP transport is included with the ORB. (Please also refer Chapter 7 of this guide for additional information.

6.1.1.1 Transport Layers

The General Inter-ORB Protocol (GIOP) is a messaging layer that provides interoperability between CORBA applications by defining a standard way of transferring data between ORBs running on different machines.

The Common Data Representation (CDR) definition is an element of the GIOP. CDR is a transfer syntax mapping OMG IDL data types into a low level representation for *on-the-wire* transfer between ORBs.

The transport layer plays the role of transferring GIOP messages and managing connections between ORBs. TCP/IP is an example of a transport layer.

1 Please contact PrismTech for information on developing customised pluggable transports.

The CORBA specification compliance requires that an ORB provide an implementation of the GIOP layer over the TCP/IP transport. This more specific combination of the GIOP (messaging layer) and TCP/IP (transport layer) is known as the *Internet Inter-ORB Protocol* (IIOP).

Although TCP/IP is a very commonly used protocol, certain applications may have constraints which require them to use alternative transports. TCP/IP, for example, is a connection oriented protocol with every data packet sent over TCP/IP requiring an acknowledgement from the receiver. This is an overhead some applications cannot afford. To deal with such situations, ORB vendors often provide implementations of alternative transports. Accordingly, the OMG has defined an Extensible Transport Framework (ETF) Specification.

The Spectra ORB has gone further by providing the user with the ability to write their own custom transport plugins. In addition to supporting the TCP/IP protocol, users can use the ETF to add their own transports to the ORB to meet specific application needs.

6.2 Libraries

The ORB's runtime is distributed as a set of libraries to be linked with the user's application code. This section lists each library, explaining its purpose and the circumstances in which it should be used.

Note that this section refers to the ORB's libraries by their base names. The actual names of the libraries will be platform specific, according to the naming conventions of each platform. For example, the `e_orb` library may be called `e_orb.lib`, `libe_orb.so`, `libe_orb.a`, `libe_orb.lib`, etc. depending on the platform it is intended for.

Library	Description
`e_orb, ec_core`	The ORB core implementation
`ec_iiop`	IIOP protocol plugin (C library)
`ec_diop`	TAO DIOP protocol plugin (C library)
`ec_uiop`	TAO UIOP plugin (C library)
`e_poa`	POA implementation
`ec_tcp`	TCP transport plugin (C library)
`ec_un`	UN transport plugin (C library)
`e_any`	Any support plugin.
`e_rtorb, ec_rt`	Real-time ORB plugin.
`e_rtpoa`	Real-time POA plugin.
`e_codec`	Codec plugin
`ec_os`	Operating system portability layer (C library)

Table 3: Spectra ORB Libraries

Any Spectra ORB-based client or server application, however simple, must be linked to these libraries:

- The ORB core (`e_orb`) and shared C core library (`ec_core`)
- A C protocol (for example `ec_iiop, ec_uiop` or `ec_diop`)

- A *C* transport (for example `ec_tcp`, `ec_un` or `ec_udp`)
- The C portability layer (`ec_os`)

Server applications, i.e. applications which are used as servers, must also be linked to the POA library, `e_poa`. In addition to these mandatory libraries, link to other libraries as required to provide the functionality your application uses.

In addition to these libraries, there will be additional, platform-specific system libraries which the ORB requires.

Spectra ORB also includes the plugins listed below. Please note that to use any of these plugins the add method on the particular plugin class must be called before initializing the orb.

```
EORB::Plugin::Any
EORB::Plugin::Current
EORB::Plugin::POA
EORB::Plugin::RTORB
EORB::Plugin::RTPOA
EORB::Plugin::Codec
```

6.2.1 Library Definitions

e_orb Library

The `e_orb` library contains the core components of the ORB, such as `CORBA::ORB` and `CORBA::Object`. All ORB-based applications must link with this library.

ec_core

The `ec_core` library contains common ORB components which are shared with the C ORB implementation.

ec_iiop Library

The `ec_iiop` library contains the implementation of a pluggable protocol, implementing IIOP.

ec_tcp Library

The `ec_tcp` library contains an implementation of the transport parts corresponding with the `ec_iiop` library.

ec_diop Library

The `ec_diop` library contains the implementation of a pluggable protocol, implementing DIOP.

ec_udp Library

The `ec_udp` library contains an implementation of the transport parts corresponding with the `ec_diop` library.

e_any Library

The `e_any` library contains the implementation for the pluggable `CORBA::Any` functionality. It also contains the full implementation for `CORBA::TypeCode` (which only has a minimal implementation in the `e_orb` library).

e_poa Library

i If your application is acting purely as a CORBA client, then this library is not required, enabling a significant reduction in footprint. However, server applications *must* link to the `e_poa` library. The `e_poa` library contains the implementation of the pluggable ORB's Minimum CORBA POA, and related classes. This enables an application to activate and service requests on objects.

This library also contains the implementation for `PortableServer::Current`, which is also pluggable.

ec_os Library

The `ec_os` library contains ORB internal, platform-specific code. This library serves as an abstraction layer for the other ORB libraries, therefore it must be linked in all ORB applications.

ec_un Library

The `ec_un` library contains an implementation of the unix domain socket transport required by the UIOP protocol.

ec_uiop Library

The `ec_uiop` library contains the implementation of a pluggable UIOP (Unix domain sockets) protocol implementation.

e_codec Library

The `e_codec` library contains the implementation of a pluggable `IOP::Codec` module for the manipulation of CDR data encapsulations.

e_rtorb

The `e_rtorb` library contains the components of the realtime ORB.

e_rtpoa

The `e_rtpoa` library contains the components of the realtime POA.

ec_rt

The `ec_rt` library contains common realtime ORB components which are shared with the C ORB implementation.

6.3 _this () Method Behaviour

All servant skeleton classes provide an `_this()` method. The full CORBA specification states that the method has three purposes.

Purpose One - Within the context of a request invocation on the target object represented by the servant, it allows the servant to obtain the object reference for the target CORBA object it is incarnating for that request. This is true even if the servant incarnates multiple CORBA objects. In this context, `_this()` can be called regardless of the policies used to create the dispatching POA.

In this context, `_this()` relies on the functionality offered by `PortableServer::Current` to obtain information about the target of the current invocation. Since the ORB's implementation of `PortableServer::Current` is pluggable, the above description may or may not apply, depending on whether `PortableServer::Current` is plugged in or not. If it is not plugged in, then `_this()` will behave as if it is being called outside the context of an operation invocation.

Purpose Two - Outside the context of a request invocation, on the target object represented by the servant, it allows a servant to be implicitly activated if its POA allows implicit activation. This requires the activating POA to have been created with the `IMPLICIT_ACTIVATION` policy. If the POA was not created with the `IMPLICIT_ACTIVATION` policy, the `PortableServer::WrongPolicy` exception is thrown. The POA used for implicit activation is obtained by invoking `_default_POA()` on the servant.

Since the `IMPLICIT_ACTIVATION` policy is not by Spectra ORB, the above should be read, assuming that the `NO_IMPLICIT_ACTIVATION` policy is in effect, that is if the servant is not active, then a `PortableServer::WrongPolicy` exception will be thrown.

If a servant has been activated in a child POA, it is advisable that the servant should override the `_default_POA()` method to return that POA in which it is active. This ensures that when the `_this()` method attempts to lookup the servant, or create a reference for the servant, it does so using the correct POA, instead of the root POA.

Purpose Three - Outside the context of a request invocation on the target object represented by the servant, it will return the object reference for a servant that has already been activated, as long as the servant is not incarnating multiple CORBA objects. This requires the POA with which the servant was activated to have been created with the `UNIQUE_ID` and `RETAIN` policies. If the POA was created with the `MULTIPLE_ID` or `NON_RETAIN` policies, the `PortableServer::WrongPolicy` exception is thrown. The POA is obtained by invoking `_default_POA()` on the servant.

Again, it is important to note that the `_default_POA()` method on the servant be overridden in order to return the appropriate POA, otherwise the above behaviour cannot be achieved.

6.4 Exceptions

6.4.1 Platforms Supporting C++ Exceptions

The ORB supports standard C++ exception handling on platforms that support exception handling for both ORB library operations and operations on objects, provided the appropriate build has been used for your ORB distribution.

In the standard case, an exception may be thrown from an object implementation in a server and caught by the calling client using the standard C++ exception-handling syntax.

Example 1 Standard C++ Exception Handling

```
// idl
exception myException
{
   long code;
};

interface myInterface
{
   void op (in long val) throws myException;
};


// client
try
{
   myIp->op (101);
}
catch (const myException& me)
{
   cerr << "caught myException: " << me.code;
}
catch (const CORBA::SystemException& se)
{
   cerr << "caught CORBA::SystemException: "<< se._rep_id () << ": "
      << se.minor ) << endl;
}
catch (...)
{
   cerr << "caught some other exception" << endl;
}


// server
void myInterface_impl::op(CORBA::Long val)
{
   if (val > 100)
   {
      myException me (val);
      throw me;
   }
}
```

6.4.2 Platforms Without C++ Exception Support

The C++ Language Mapping specification provides a `CORBA::Environment` class and an alternative method signature for passing exception information for platforms that do not support C++ exceptions.

A `CORBA::Environment` parameter is passed to each operation and is used to convey exception information to the caller for platforms which do not support exception handling. The `CORBA::Environment` variable may be stack- or heap-allocated.

The `CORBA::Environment` class holds an exception of type `CORBA::Exception` and provides methods to access and modify that exception.

Both user-defined and system exceptions are derived from `CORBA::Exception`. This allows exceptions to be caught either by their actual type, by their generic type (`CORBA::UserException` or `CORBA::SystemException`), or by their base type (`CORBA::Exception`).

When an exception is caught by either its base type or its generic type, its actual type can be determined by dynamic cast. On C++ platforms that do not support Run Time Type Information (RTTI), the standard provides a static `_downcast()` method for dynamic casting. Alternatively, the exception's repository ID may be obtained by calling the exception's `_rep_id()` method.

! Object implementers must take special care to ensure that when an exception is thrown on a non-exception-handling platform, all out and return pointers are returned to the caller as null pointers. An uninitialized pointer results in undefined behaviour.

Example 2 Without Native C++ Exception Handling

The exception handling shown in *Example 1* above can be implemented like this:

```
// client
CORBA::Environment env;
CORBA::Exception * exc;
CORBA::SystemException * sep;

myIp->op (42, env);

if (exc = env.exception ())
{
   myException * mep = myException::_downcast(exc);
   if (mep)
   {
      cerr << "caught myException: " << mep->code << endl;
   }
   else if (sep = CORBA::SystemException::_downcast (exc))
   {
      cerr << "caught CORBA::SystemException: "
      << se._rep_id () << ": "
      << se.minor () << endl;
   }
   else
   {
      cerr << "caught some other exception" << endl;
   }
}

// server
void myInterface_impl::op (CORBA::Long val, CORBA::Environment & env)
{
   if (val > 100)
   {
      env.exception (new myException (100));
   }
}
```

6.4.3 Portable Exception Layer Macros

6.4.3.1 Description

The Portable Exception Layer enables seamless cross-platform programming by providing C++-style exception handling on platforms where it is not supported. Code written with this layer works on platforms that support true exception handling as well as platforms that do not. Code will be compliant with the C++ mapping, on either true-exception or non-exception platforms, when these macros are used.

Files or modules which use the macros must include the following header file: `eOrb/Corba/Environment.h`

Example 3 Using Portable Exception Layer Macros

The previous examples can be implemented using the macros in the example code shown below.

```
// client
EORB_DECLARE_ENV;

EORB_TRY
{
   myIp->op (101, EORB_ENV_VARN);
   EORB_CHECK_ENV;
}
EORB_CATCH (myException, me)
{
   cerr << "caught myException: " << me.code;
}
EORB_CATCH (CORBA::SystemException, se)
{
   cerr << "caught CORBA::SystemException: "
   << se._rep_id () << ": << se.minor( ) << endl;
}
EORB_CATCH_ANY ()
{
   cerr << "caught some other exception" << endl;
}
EORB_END_TRY


// server
void myInterface_impl::op (CORBA::Long val EORB_ENV_ARGN)
{
   if (val > 100)
   {
      myException me (val);
      EORB_THROW_RETURN_VOID (me);
   }
}
```

6.4.3.2 Macro Definitions

EORB_DECLARE_ENV

Declares a local `CORBA::Environment` instance on the stack, for use by the other Portable Exception Layer macros. If building with exceptions, this expands to nothing.

If a function uses any of the Portable Exception Layer macros, it must either have an `EORB_ENV_ARG[1|N]` in its signature, or it must use `EORB_DECLARE_ENV` in its body.

In the following example, `EORB_DECLARE_ENV` is required to support the correct operation of the `EORB_TRY/CHECK/CATCH` macros.

```
int main (int argc, char **argv)
{
   EORB_DECLARE_ENV;

   EORB_TRY
   {
      do_something (EORB_ENV_VAR1);
      EORB_CHECK_ENV;
   }
   EORB_CATCH (CORBA::Exception, e)
   {
      cerr << "Caught: " << e->_rep_id() << endl;
   }
   EORB_END_TRY
   …
}
```

In the next example, `EORB_DECLARE_ENV` is not required because the `CORBA::Environment` is passed into the function by declaring it with `EORB_ENV_ARG1` in its signature. In this case, it is the responsibility of the caller to create, and pass in the `CORBA::Environment`.

```
void func (EORB_ENV_ARG1)
{
   do_something (EORB_ENV_VAR1);
   EORB_CHECK_ENV_RETURN_VOID;

   do_something_else (EORB_ENV_VAR1);
   EORB_CHECK_ENV_RETURN_VOID;
}
```

EORB_ENV_ARG[1 | N]

Declares a `CORBA::Environment` formal argument in the signature of a function.

The two forms, ending in either `1` or `N`, are for use where the signature has no other arguments, or when it does have other arguments, respectively.

```
void func1 (EORB_ENV_ARG1)
{
   …
}
```

```
void func2 (CORBA::Short arg1 EORB_ENV_ARGN)
{
   …
}
```

Note that in the signature of `func2`, there is no comma between the last argument and the `EORB_ENV_ARGN` macro. The macro expansion contains the comma, in the non-exception case, otherwise the macro expands to nothing.

EORB_ENV_VAR[1 | N]

Passes the current `CORBA::Environment` as an actual argument to a function call.

The two forms, ending in either `1` or `N`, are for use where the signature of the function being called has no other arguments, or when it does have other arguments, respectively.

The actual `CORBA::Environment` passed by this macro is picked up from the local scope, using a default name. This local may have been declared within the local scope, using `EORB_DECLARE_ENV`, or passed into the local scope (function) using `EORB_ENV_ARG[1|N]`.

The following example shows the use of `EORB_ENV_VAR[1|N]`:

```
void func0 (EORB_ENV_ARG1)
{
   …
}

void func1 (int arg EORB_ENV_ARGN)
{
   …
}

void func1 (EORB_ENV_ARG1)
{
   func0 (EORB_ENV_VAR1);
   EORB_CHECK_ENV_RETURN_VOID;

   func1 (1 EORB_ENV_VARN);
   EORB_CHECK_ENV_RETURN_VOID;
   …
}
```

EORB_TRY

Begins a try-catch block.

EORB_TRY_LABEL(label)

Begins a try-catch block with a label to facilitate multiple try-catch blocks within a single function.

EORB_THROW (exc)

Throws the specified exception, within the context of a try block.

In true exception handling environments, this macro expands to a simple throw. In a non exception handling environment, it puts the specified exception into the current `CORBA::Environment` and causes execution to jump immediately to check for and enter an appropriate `EORB_CATCH` block. If an appropriate `EORB_CATCH` block cannot be found, execution will continue following the `EORB_END_TRY`.

This macro can only be used within an `EORB_TRY/CATCH` construct.

```
CORBA::Boolean some_func (EORB_ENV_ARG1)
{
   EORB_TRY
   {
      …
      EORB_THROW (CORBA::IMP_LIMIT (0, CORBA::COMPLETED_NO));
      …
   }
   EORB_CATCH (CORBA::Exception, e)
   {
      return 0;
   }
   EORB_END_TRY

   EORB_CHECK_ENV_RETURN (0);

   return 1;
}
```

i Note the use of an `EORB_CHECK_ENV_RETURN(0)` immediately following the `EORB_END_TRY`. This ensures that if code within the `EORB_TRY` block throws an exception that is not caught, execution will return immediately to the caller.

EORB_THROW_LABEL (exc, label)

Throws the specified exception, within the context of an `EORB_TRY_LABEL/CATCH` construct.

This macro is identical to `EORB_THROW(exc)` except that it should be used when in the context of a labelled try-catch block i.e. one which begins with `EORB_TRY_LABEL(label)`. Labelled try-catch blocks are required when more than one try-catch block is used within the same function.

```
void some_func (EORB_ENV_ARG1)
{
   EORB_TRY
   {
      EORB_TRY_LABEL(a)
      {
         EORB_THROW_LABEL (MyException(), a);
      }
      EORB_CATCH (CORBA::Exception, e)
      {
         cerr << e._rep_id() << endl;
      }
      EORB_END_TRY
   }
   EORB_CATCH (CORBA::Exception, e)
   {
      cerr << e._rep_id() << endl;
   }
   EORB_END_TRY
}
```

EORB_THROW_RETURN (exc, ret)

Throws the specified exception, when not in the context of an `EORB_TRY/CATCH` construct i.e. where throwing the exception should cause the current function to exit immediately to the caller.

The `ret` argument specifies the return value of the function, in the case of a non-exception build.

This macro should only be used outside of `EORB_TRY/CATCH` constructs. For example:

```
CORBA::Boolean some_func (EORB_ENV_ARG1)
{
   …
   EORB_THROW_RETURN (MyException(), 0);
   …

   return 1;
}
```

EORB_THROW_RETURN_VOID (exc)

Throws the specified exception, when not in the context of an `EORB_TRY/CATCH` construct i.e. where throwing the exception should cause the current function to exit immediately to the caller.

This macro is identical to `EORB_THROW_RETURN(exc,ret)` except no return value is involved because it is for use in functions having a void return type.

```
void some_func (EORB_ENV_ARG1)
{
   …
   EORB_THROW_RETURN (MyException());
   …
}
```

EORB_CATCH(type,var)

Catches an exception of the specified type whose instance name is specified by var.

EORB_CATCH_NOOP(type)

Catches an exception of the specified type but does not use the exception instance.

By not declaring a local variable to hold the exception, this avoids compiler warnings about 'unused local variables'.

EORB_CATCH_ANY

Catches any exception.

EORB_RETHROW_RETURN(ret)

Re-throws the exception, from within the context of a catch block, and returns ret in non-exception handling environments.

```
CORBA::Boolean func (EORB_ENV_ARG1)
{
   EORB_TRY
   {
      another_func (EORB_ENV_VAR1);
      EORB_CHECK_ENV;
   }
   EORB_CATCH (CORBA::Exception, e)
   {
      EORB_RETHROW_RETURN (0);
   }
   EORB_END_TRY

   return 1;
}
```

EORB_RETHROW_RETURN_VOID

Re-throws the exception, from within the context of a catch block.

```
void func (EORB_ENV_ARG1)
{
   EORB_TRY
   {
      another_func (EORB_ENV_VAR1);
      EORB_CHECK_ENV;
   }
   EORB_CATCH (CORBA::Exception, e)
   {
      EORB_RETHROW_RETURN_VOID;
   }
   EORB_END_TRY
}
```

EORB_CHECK_ENV

This macro is for use within the context of a try block. Within a try block, each statement which may raise an exception must be immediately followed by an `EORB_CHECK_ENV`.

In non exception handling environments, this is the mechanism that transfers control to the appropriate catch block if an exception is 'thrown'.

In a true exception handling environment, this macro expands to nothing.

```
void func()
{
   EORB_DECLARE_ENV;

   EORB_TRY
   {
      func1 (EORB_ENV_VAR1);
      EORB_CHECK_ENV;

      func2 (EORB_ENV_VAR1);
      EORB_CHECK_ENV;
   }
   EORB_CATCH_ANY
   {
      cerr << "ERROR" << endl;
   }
   EORB_END_TRY
}
```

EORB_CHECK_ENV_LABEL (label)

This macro is identical to `EORB_CHECK_ENV` except it is for use within a labelled `try` block *i.e.* one which begins with `EORB_TRY_LABEL(label)`.

```
void func()
{
   EORB_DECLARE_ENV;

   EORB_TRY
   {
      …
   }
   EORB_CATCH_ANY
   {
      …
   }
   EORB_END_TRY

   EORB_TRY_LABEL (second)
   {
      func1 (EORB_ENV_VAR1);
      EORB_CHECK_ENV_LABEL (second);
   }
   EORB_CATCH_ANY
   {
      cerr << "ERROR" << endl;
   }
   EORB_END_TRY
}
```

EORB_CHECK_ENV_RETURN (ret)

This macro is similar to EORB_CHECK_ENV, except it is for use outside the context of a try block *i.e.* where an exception would cause execution to transfer immediately out of the function, up to its caller.

In non-exception handling environments, this macro simulates stack unwinding by using a return statement. The `ret` argument of this macro should evaluate to a valid return value for the enclosing function purely to satisfy the compiler. The assumption is that the caller will not actually use this value because of the exception.

```
CORBA::Long func (EORB_ENV_ARG1)
{
   func1 (EORB_ENV_VAR1);
   EORB_CHECK_ENV_RETURN(0);

   func2 (EORB_ENV_VAR1);
   EORB_CHECK_ENV_RETURN(0);

   return 32;
}
```

EORB_CHECK_ENV_RETURN_VOID

This is identical to the `EORB_CHECK_ENV_RETURN(ret)` macro except it is for use in functions that have a void return type.

```
void func (EORB_ENV_ARG1)
{
   func1 (EORB_ENV_VAR1);
   EORB_CHECK_ENV_RETURN_VOID;

   func2 (EORB_ENV_VAR1);
   EORB_CHECK_ENV_RETURN_VOID;
}
```

EORB_END_TRY

Declares the end of a try-catch block. This should be placed after the closing bracket of the last catch block.

6.4.4 Minor Exception Codes

All CORBA system exceptions have a minor code, that can be obtained with the `minor()` operation. These give additional detail as to why the exception was raised and can help narrow down thwe cause of the problem if raised with the PrismTech support organisation. Minor codes are documented in the `include/eOrbC/EORB/minor.h` header file in the ORB distribution. Note that minor codes are an unsigned 32 bit integer, the bottom most 16 bits being used to identify the minor code and the top 16 bits being used to identify the vendor.

6.5 Multiple Profiles

The ORB supports a number of pluggable profiles (IIOP, UIOP, DIOP and others).

By default each installed profile will create a listener and multi-profile IORs will be generated. A client can control which profile is used by either using a client side policy (`RTCORBA::Protocol`) or setting the default policy to use via the ORB initialization argument *`-ORBProtocol <protocol>`*.

6.6 Logging

The debug build of the ORB support logging using the *`-ORBLogLevel <level>`* ORB initialization argument. The available logging levels are Debug, Stats, Warn, Error and Fatal. Figure 7 shows the breakdown of diagnostic messages with thread information included.

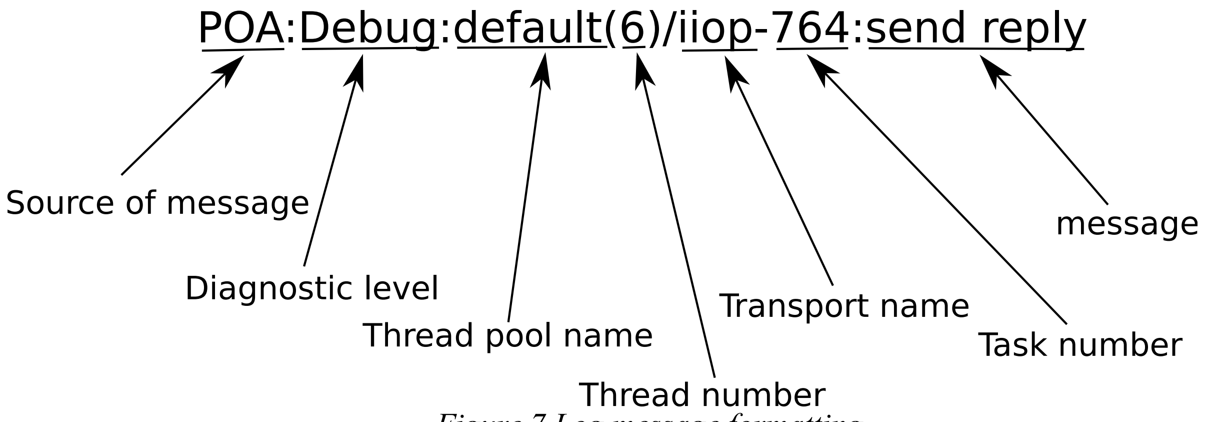


Figure 7 Log message formatting

7 Extensible Transport Framework

7.1 What is the Extensible Transport Framework?

The Extensible Transport Framework is an Object Management Group (OMG) specification for a pluggable ORB transport architecture. The purpose of the ETF specification is to provide a high level of abstraction between the transport and ORB messaging layer allowing development of interoperable transport plugins. IIOP (GIOP over TCP/IP) imposes limitations in the form of unpredictable latencies and communication overheads unsuitable for many embedded and real-time systems.

ETF aims to separate the concerns of the messaging layer (GIOP) with that of the transport layer thus facilitating independent development of alternative transports (for example, TCP) without the need for a custom messaging layer. Furthermore the ETF specification defines a standard object oriented approach to transport functionality and interaction with the ORB greatly simplifying the process of 3rd party transport development.

An ETF transport can be viewed at the highest level as a collection of objects each of which is responsible for representing a particular subset of a transport functionality. The following objects are described by the ETF specification:

- `Factories`
- `ProtocolProperties`
- `Profile`
- `Connection`
- `Listener`

7.1.1 Factories

The Factories object is the entry point through which the ORB can gain access to the underlying transport. The primary purpose of the Factories object is to create other ETF objects on demand such as Connection and Listener objects in addition to providing information about the transport such as an identifier and version number.

7.1.2 Protocol Properties

The purpose of the `ProtocolProperties` object is to represent attributes and customisable properties for a transport plug-in. The transport attributes held by a `ProtocolProperties` object usually concern the performance and/or behaviour of the transport. As an example, packet size or send and receive window size are typical attributes that might be used for a TCP transport plug-in.

7.1.3 Profile

The Profile object holds information about the server endpoint. This information is required by a client to be able to connect to the server. In a typical example of a TCP/IP transport, this information would usually be a host name and port number.

7.1.4 Connection

The Connection object holds state information regarding a client's connection to a server and or vice versa and provides operations to establish, interact with and shut down a connection.

7.1.5 Listener

A Listener object is associated with a server endpoint and is responsible for accepting incoming connection requests and establishing and managing server side connections.

7.2 Developing a Custom Transport

All transport implementations are written in C and are shared with that of the C version of the ORB. For details of how to implement a custom transport see Chapter 7 of the Spectra ORB C Edition User Guide.

8 Using the ORB

8.1 Introduction

The major steps of creating a CORBA-based client-server application using Spectra ORB are.

Step 1: Declare the application's classes and/or interfaces in IDL. (Refer to the *Using the IDL Compiler* section of the *IDL Guide* for *Step 1* through *Step 4*.)

Step 2: Compile the IDL in order to create C++ sources for the stubs, skeletons and/or tie classes and interfaces.

Step 3: Write the application's server modules (if they are not being implemented by third parties). These are derived from the generated IDL *server implementation base* header files (containing the *skeleton* and/or *tie* declarations) and must include (using the `#include` directive) the servant implementation base file (an IDL generated file with an `_s` suffix).

Step 4: Write the application's client modules (if they are not being implemented by third parties). These modules must include (using the `#include` directive) the generated IDL client header file (containing the *stub* declarations).

Step 5: Compile the developer-written source code and IDL-generated source code with the C++ compiler for the required platform, ensuring that the compiled code is linked to the ORB's libraries for that specific platform.

Step 6: Deploy the application's server and client components on the required platforms.

Step 7: Start the server and run the client.

This section describes the procedures, requirements and practical details needed to create CORBA-based applications with Spectra ORB. (Please note that this section is ***not*** intended as a tutorial of how to write CORBA-based applications.) The topics covered in this section include:

- a general description of how to use the Spectra ORB IDL compiler, `idlcpp`
- the requirements, procedures and settings for creating, compiling, linking, deploying and running applications which are common to all platforms supported by Spectra ORB
- the particular information needed to compile, link, deploy and run the developer-written and IDL-generated source code for each, specific platform supported by Spectra ORB

8.1.1 Conventions

The following conventions are used in this section:

`$EORBHOME` or `%EORBHOME%`	- is the directory where Spectra ORB is installed
`$EORBENV` or `%EORBENV%`	- is the target platform type, e.g. `linux-gcc-x86`

8.2 Using the IDL Compiler

Basic instructions for using Spectra ORB's IDL to C++ compiler, `idlcpp`, are given here.

Detailed information, including descriptions of all of the compiler's command line options are provided in the *IDL Guide*. Special instructions for specific platforms, such as environment variable settings, are provided in the *Product and Installation Guide*.

! You should read the instructions provided in both the *IDL Guide* and *Product and Installation Guide* before attempting to use the IDL compiler, `idlcpp`.

The idlcpp compiler must be in the system's `PATH` in order to run. The `idlcpp` compiler is used from the command line as:

```
% idlcpp [options] <idl_files>
```

where

`[options]`	is a list of zero or more command-line options
`<idl_files>`	is a list of one or more IDL source files

The IDL source files must have `idl` as the filename extension, for example `myfile.idl`.

Using `idlcpp` with no parameters or with the -u option displays usage information. The complete list of command-line parameters is described in the IDL Guide: please refer to the instructions in the *IDL Guide* before using the `idlcpp` compiler.

The IDL compiler's default behaviour is to create two C++ source files and two C++ header files for each IDL file. The number, type and names of the generated output files depend on the command-line options used.

The standard generated source files used by clients are:

- a header file which has a `.h` extension, *e.g.* `hello.h`; this file contains stub declarations
- an implementation file which has a `.cpp` extension, *e.g.* `hello.cpp`; this file contains stub definitions

The standard generated source files used by servers are:

- a header file which has an `_s` suffix and `.h` extension, *e.g.* `hello_s.h`; this file contains skeleton and optional tie declarations
- an implementation file which has an `_s` suffix and `.cpp` extension, *e.g.* `hello_s.cpp`. This file contains skeleton definitions.

Refer to the *IDL Guide* for the appropriate command line options to use to generate specific files.

Example

To generate the client and server stub and skeleton files from an IDL source file called `hello.idl` use:

```
% idlcpp hello.idl
```

This will generate:

- `hello.h` and `hello.cpp`, the C++ client header and implementation files (containing the stubs)

- `hello_s.h` and `hello_s.cpp`, the C++ server header and implementing files (containing the skeletons).

8.3 Requirements Common to All Platforms

This section describes the requirements, procedures and settings for creating, compiling, linking, deploying and running applications common to all platforms supported by Spectra ORB. Information specific to each supported platform is provided under Section 6.4, *Application Creation Example.*

Pre-built libraries for each supported platform are in the `$EORBHOME/lib/<platform>` directory. The Spectra ORB product includes examples which can be used to verify that Spectra ORB and your host or target platform is correctly installed and configured.

An application's source file modules (*i.e.* the C++ coded files) may need to include the ORB's libraries that are listed in Section 4.2, *Libraries*.

Developers must determine which methodologies their application clients will use to find or resolve servers, including their own application servers and other, third party servers (such as the *Naming Service*). Refer to section *8.3.2.1, Resolving Servers, on page 54,* for a list of alternative resolution approaches.

i Spectra ORB licensing information pertaining to all platforms is provided in the *Product and Installation Guide*.

8.3.1 Compiling and Linking Applications

The procedures which are common to all platforms and compilers for compiling and linking the source files (including both the IDL generated C++ files and developer-written C++ implementations) are described in this section.

The following requirements are common to all platforms, unless otherwise stated.

- `$EORBHOME/include` must be added to the compiler's *includes* search path.
- `$EORBHOME/lib/$EORBENV` must be added to the linker's library search path

The developer-written client and server application files must be linked with the correct object files. The object files will have been compiled from the application's IDL-generated sources.

It is natural and correct to assume that the *client*-side files must be linked with object files compiled from the IDL-generated *client* files. However, the CORBA architecture and specification requires that the server-side files must also be linked with the client files, not only with the equivalent server object files. Refer to *The Common Object Request Broker: Architecture and Specification* from the OMG.

Example

An IDL specification file, `hello.idl`, is used to generate C++ client and server files, including:

`hello.h` and `hello.cpp` for the client

`hello_s.h`, and `hello_s.cpp` for the server

The C++ compiler will create the following object files from these files:

`hello.o` for the client

`hello_s.o` for the server

The developer-written client-side files must be linked with `hello.o`.

The developer-written server-side files must be linked with `hello_s.o` and with `hello.o`.

8.3.1.1 Required Libraries

The Spectra ORB libraries are described under Section 6.2, *Libraries*. Spectra ORB-based client or server applications must be linked to:

- the Spectra ORB core (`e_orb`, `ec_core`)
- a protocol (`ec_iiop`)
- a transport (`ec_tcp`)
- the platform abstraction library (`ec_os`)

! Server applications, *i.e.* applications which are used as servers, must also be linked to the POA library, `e_poa`.

8.3.1.2 Build Version Options

Spectra ORB-based applications can be built as different versions, including those which:

- contain or do not contain debug information
- either have or do not have native C++ exception handling[1]

! The *Debug vs Release*, *Exception Handling and Run-Time Type Information*, and platform-specific compiler and linker sections provide essential information required for successfully compiling your application.

8.3.1.2.1 Debug vs Release

Each ORB distribution comes with a set of debug and release libraries. The debug library names have a `_g` suffix to distinguish them from the standard release libraries. For example, the *standard* release version of the `e_orb` library is called `e_orb.dll`; the *debug* version is called `e_orb_g.dll`.

The `-DE_DEBUG` switch must be used with the compiler if the compiled code is to be linked against the debug version of the ORB libraries since there is inline code that is conditional on the `E_DEBUG` symbol.

The compiler and linker switches that some compilers and/or platforms must use depend on their particular ORB distribution's configuration.

8.3.1.2.2 Exception Handling and Run-Time Type Information

The ORB is distributed with or without support for true C++ exceptions. Which type of distribution you have will depend on your requirements, but the version will also affect how your application code must be compiled.

1 A particular Spectra ORB build will either have native C++ exception handling or it will not have native C++ exception handling.

8.3.2 Deploying and Running Applications

Information that is common to all Spectra ORB supported platforms about running applications is described here. The aspects and procedures specific to particular platforms are described under *Application Creation*.

Regardless of whether clients and servers are run from the same or different machines or any particular platform, they always:

- Run as separate processes. Clients and servers are started in their own, separate shells, windows or processes.
- Must be able to locate each other. Clients locate servers using one of the methods described under *Resolving Servers* below. Servers locate clients using the internal mechanisms provided by the ORB.

Also, note that servers can change their connection protocol, host or port values by using the `-ORBListenEndpoints` command line option. This option automatically sets the server 's IOR to the values used with `-ORBListenEndpoints` and is described under *Server Listening Point* below.

8.3.2.1 Resolving Servers

An application's clients and server are run as separate processes. Subject to the limitations of particular platforms, developers can implement their client(s) so that they can find or *resolve* their server by:

- reading the server's IOR from a file created by the server, *or*
- using a corbaloc URL (this URL is output to the screen when the server is first invoked), *or*
- using the Naming Service.

8.3.2.1.1 Server Listening Point

A server's Root POA can be instructed to listen with a specified network protocol on a specified port, referred to here as its *endpoint*, at run time by using the `-ORBListenEndpoints` option with the server's executable.

The `-ORBListenEndpoints` option is a list of comma-separated endpoints as described by the following EBNF:

```
endpoints = endpoint | endpoint "," endpoints
endpoint = iiop_endpoint | diop_endpoint | uiop_endpoint
iiop_endpoint = "iiop:" [host] ":" [port]
diop_endpoint = "diop:" [host] ":" [port]
uiop_endpoint = "uiop:" [filename]
host := host_name | ip_address
```

Supplying an `-ORBListenEndpoints` argument will disable the use of default endpoints such that only the specified endpoints will be used by the Root POA.

Example 1: Complete IIOP

This example shows the command line for setting all the values with IIOP for the Root POA in a server called `server`.

```
% server -ORBListenEndpoints iiop:213.48.91.157:8001
```

Example 2: IIOP, Root POA and All Interfaces

This example is the same as *Example 1*, except that the optional `host` value is also omitted, thereby specifying the port to be used by the Root POA. The default host name will be used.

```
% server -ORBListenEndpoints iiop::8001
```

Example 3: Multiple Endpoints

This example shows using two endpoints to listen on both IIOP and UIOP. Default values for IIOP host and port and a default filename for UIOP will be used.

```
% server -ORBListenEndpoints iiop:213.48.91.157:8001,uiop:/tmp/ep
```

8.4 Platform-Specific Requirements

For details on how to build and run Spectra ORB SDR C++ applications on specific platforms such as VxWorks or Integrity please refer to the *Spectra ORB Product and Installation Guide.*

8.5 Application Creation

The following procedure lists the steps required to create, compile, deploy and run an application. This is based on the simple hello example provided in `examples/cpp/hello/native`

Step 1: Declare the application's classes and/or interfaces in IDL.

Step 2: Create C++ sources by compiling the IDL.

No command line options are needed for compiling the example since it uses all of the compiler's default settings. (Both the client and server files are needed, default output file names are used and default file extensions are used.)

Step 3: Write the application's server modules.

The example server implementation file is called `server.cpp`. The module implements the IDL-generated server interfaces and methods.

This module must contain the following *`#include`* statements:

```
#include "hello_s.h" // the IDL generated skeleton
```

The server must also include a mechanism for making its objects' IORs available to clients (object resolution): this example uses `fstream` to save stringified IORs to a file; clients can then obtain the IOR from the file.

Step 4: Write the application's client modules.

The example client implementation file is called `Client.cpp`. The module implements the IDL-generated client interfaces and methods.

This module must contain the following `#include` statements:

```
#include "hello.h" // the IDL generated stub
```

The client must also include a mechanism for retrieving the server's IORs (object resolution): this example uses `fstream` to read stringified IORs from a file created by the server.

Step 5: Compile and link the C++ source code.

8.5.1 Build Directives

The example is being built as a *release* build with *native exception* support.

UNIX No additional directives are required other than using the libraries as listed under *Linking* below.

WIN Add the `/GX` and `/GR` switches to the Microsoft Visual C++ Project's Project options box located on the C/C++ tab of the Project Setting dialogue pane before compiling. Developers should refer to the appropriate SDK documentation.

8.5.2 Compiling

The example server and client modules are compiled with the following ORB-specific include and library directories:

```
$EORBHOME/include
$EORBHOME/lib/$EORBENV
```

Examples

UNIX A Linux Intel x86 host/target build using the `g++` compiler would use:

```
% g++ -c -I$EORBHOME/include -L$EORBHOME/lib/linux-gcc-x86 server.cpp
```

WIN Using the Microsoft Visual C++ compiler for a Windows host/target, add `win32_msdev_x86` to the Preprocessor definitions box located on the C/C++ tab of the Project Setting dialogue pane before compiling.

8.5.3 Linking

The example server and client modules are linked with the following ORB-specific libraries:

`ec_iiop, ec_tcp, e_orb, ec_core, and ec_os`

The server must also be linked with the server *and* client object files, `hello_s.o` and `hello.o` in this example.

UNIX A Linux Intel x86 host/target build using the `g++` compiler would use:

```
% g++ -o server hello.o hello_s.o -L$EORBHOME/lib/linux-gcc-x86 \
     -lec_iiop -lec_tcp -le_orb -lec_core -lec_os
```

Using the Microsoft Visual C++ compiler (or the Microsoft eMbedded Visual C++ compiler for WinCE) for a Windows host/target, add the libraries listed above (*e.g.* `e_orb.lib`, etc.) to the Object/libraries modules box located on the Link tab of the Project Setting dialogue pane before compiling.

Step 6: Deploy the application.

Copy the compiled executable files to the directories where they are intended to be run from. The server and client executables for this example have been designed to be run from the same directory.

Step 7: Start the server and run the client.

For this example, create separate shells or windows for the server and client: first run the server, then the client. The client obtains example objects from the server, then prints each object's name.

UNIX Run `server`, then run `client`, each in separate xterm windows.

WIN Run `server.exe`, then run `client.exe`, each in separate DOS windows.

Run the server and client either from the IDE or each in separate DOS windows on the target device.

The server prints its IOR that can be passed to the client as an argument, allowing the client to resolve the server reference.

```
./server
Hello native server starting
-ORBInitRef
hello=IOR:010000001800000049444c3a4772656574696e67536572766963653a312e300
00100000000000003c000000010102000a00000031302e312e302e3238003bce20000000
01921800654f5242bb916e3100000000005454f52420000000400000000000000000000000
```

8.6 Example Source Code

A set of C++ examples are provided in `examples/cpp` along with html instructions for compiling and running them. To determine the exact flags used to compile the examples GNU make can be executed with verbose output enabled using:

```
gmake VERBOSE=
```

This generates output showing how the example is build eg:

```
Compiling hello.cpp
g++ -Dlinux_gcc_x86 -D_GNU_SOURCE -I/home/steve/projects/eorb/include -I.
-I/home/steve/projects/eorb/build/include -I/usr/include/nptl -c -ansi -pthread -pipe
-Wall -fno-strict-aliasing -march=i686  -O2  hello.cpp -o .bld-linux-gcc-x86/hello.o
Compiling hello_s.cpp
g++ -Dlinux_gcc_x86 -D_GNU_SOURCE -I/home/steve/projects/eorb/include -I.
-I/home/steve/projects/eorb/build/include -I/usr/include/nptl -c -ansi -pthread -pipe
-Wall -fno-strict-aliasing -march=i686  -O2  hello_s.cpp -o .bld-linux-gcc-
x86/hello_s.o
Compiling server.cpp
g++ -Dlinux_gcc_x86 -D_GNU_SOURCE -I/home/steve/projects/eorb/include -I.
-I/home/steve/projects/eorb/build/include -I/usr/include/nptl -c -ansi -pthread -pipe
-Wall -fno-strict-aliasing -march=i686  -O2  server.cpp -o .bld-linux-gcc-
x86/server.o
Compiling hello_i.cpp
g++ -Dlinux_gcc_x86 -D_GNU_SOURCE -I/home/steve/projects/eorb/include -I.
-I/home/steve/projects/eorb/build/include -I/usr/include/nptl -c -ansi -pthread -pipe
-Wall -fno-strict-aliasing -march=i686  -O2  hello_i.cpp -o .bld-linux-gcc-
x86/hello_i.o
Linking ./server
g++ -L/home/steve/projects/eorb/lib/linux-gcc-x86   -L/usr/lib/nptl  .bld-linux-gcc-
x86/hello.o .bld-linux-gcc-x86/hello_s.o .bld-linux-gcc-x86/server.o .bld-linux-gcc-
x86/hello_i.o -lec_iiop -lec_tcp -le_poa -le_orb -lec_core -lec_os   -Wl,-Bdynamic
-ldl -lpthread -lrt -o ./server
strip ./server
```

9 Bibliography

The documents listed here may provide useful information or help for CORBA users and developers.

[1] *Advanced CORBA Programming with C++*, Mich Henning and Steve Vinoski, Addison-Wesley, 1999, http://www.awprofessional.com.

[2] *The Common Object Request Broker: Architecture and Specification Revision 2.3*, June 1999, Object Management Group (OMG), http://www.omg.org.

[3] *The Common Object Request Broker: Architecture and Specification*, *Revision 3.03*, March 2004, Object Management Group (OMG), http://www.omg.org.

[4] *Minimum CORBA Specification*, *Version 1.0*, August 2002, Object Management Group (OMG), http://www.omg.org.

[5] *C++ Language Mapping Specification*, *Version 1.0*, June 2003, Object Management Group (OMG), http://www.omg.org.

[6] *Software Communications Architecture Specification, JTRS-5000 SCA V2.2.1* April 2004, http://jtrs.army.mil.

[7] *Extensible Transport Framework OMG Final Adopted Specification*, *ptc/04-03-03*, Object Management Group (OMG), http://www.omg.org/docs/ptc/04-03-03.pdf.

[8] CORBA/e Resource Page, http://www.corba.org/corba-e/

10 Appendices

10.1 Appendix A – Spectra ORB CORBA Profiles

Table 4 Provides a more detailed breakdown of the features and APIs defined in each of the CORBA Profiles supported by Spectra ORB C++ Edition. In a small number of cases the Profile may indicate that a feature is mandated but at the current time is not supported by Spectra ORB. These instances are indicated with a red tick.

Please note that the Minimum CORBA Profile does not make any explicit references to the inclusion of CORBA Services such as the Naming Service and these are listed as optional in *Table 4* . Spectra ORB C++ Edition includes implementations of Naming (Full and Lightweight), Even and Lightweight Log Services.

Scope	**Operation/ Feature**	**Minimum CORBA**	**CORBA/e Compact**	**SCA 4 Full Profile**	**SCA 4 Lightweight Profile**
IDL	Abstract Interfaces	✓	×	×	×
IDL	Value Type	✓	✓	×	×
IDL	any	✓	✓	✓	
IDL	operation context clauses	✓	×	×	×
IDL	boolean, octet, short, unsigned short, long, unsigned long, enum, float, double, long double, long long, unsigned long long, char, string	✓	✓	✓	✓
IDL	wide character string	✓	✓	×	×
IDL	unions	✓	✓	✓	✓
IDL	arrays	✓	✓	✓	✓
IDL	structs	✓	✓	✓	✓
IDL	sequence	✓	✓	✓	✓
IDL	import	✓	×	×	×
CORBA::ORB	orb_init	✓	✓	✓	✓
CORBA::ORB	id	✓	✓	×	×
CORBA::ORB	object_to_string	✓	✓	✓	✓
CORBA::ORB	string_to_object	✓	✓	✓	✓
CORBA::ORB	get_service_information	✓	✓	×	×
CORBA::ORB	list_initial_services	×	✓	×	×
CORBA::ORB	resolve_initial_references	✓	✓	✓	×
CORBA::ORB	work_pending	×	✓	✓	✓
CORBA::ORB	perform_work	×	✓	✓	✓
CORBA::ORB	run	×	✓	✓	✓
CORBA::ORB	shutdown	✓	✓	✓	✓
CORBA::ORB	destroy	✓	✓	✓	✓
CORBA::ORB	create_policy	✓	✓	✓	×
CORBA::ORB	register_value_factory	✓	✓	×	×
CORBA::ORB	unregister_value_factory	✓	✓	×	×
CORBA::ORB	lookup_value_factory	✓	✓	×	×
CORBA::ORB	register_initial_reference	✓	✓	×	×
CORBA::OBJECT	get_interface	✓	✓	×	×
CORBA::OBJECT	is_nil	✓	✓	✓	×
CORBA::OBJECT	duplicate	✓	✓	✓	×
CORBA::OBJECT	release	✓	✓	✓	×
CORBA::OBJECT	is_a	×	✓	✓	×
CORBA::OBJECT	non_existent	×	✓	✓	×
CORBA::OBJECT	is_equivalent	×	✓	✓	×
CORBA::OBJECT	hash	✓	✓	×	×
CORBA::OBJECT	get_policy	✓	✓	✓	×
CORBA::OBJECT	set_policy_overrides	✓	✓	✓	×
CORBA::OBJECT	get_client_policy	✓	✓	✓	×
CORBA::OBJECT	get_policy_overrides	✓	✓	✓	×
CORBA::OBJECT	validate_connection	✓	✓	✓	✓

Scope	Operation/ Feature	Minimum CORBA	CORBA/e Compact	SCA 4 Full Profile	SCA 4 Lightweight Profile
CORBA::OBJECT	get_orb	✓	✓	✓	×
CORBA::OBJECT	get_component	✓	×	×	×
CORBA::Policy	policy_type	✓	✓	✓	×
CORBA::Policy	copy	✓	✓	✓	×
CORBA::Policy	destroy	✓	✓	✓	×
CORBA::PolicyManager	get_policy_overrides	×	✓	✓	×
CORBA::PolicyManager	set_policy_overrides	×	✓	✓	×
CORBA::TypeCode		✓	✓	✓	×
CORBA::PolicyCurrent		×	✓	✓	×
Messaging::RebindPolicy	rebind_mode	×	✓	×	×
Messaging::SyncScopePolicy	synchonization	×	✓	✓	×
Messaging::RequestEndTimePolicy	end_time	×	✓	×	×
Messaging::ReplyEndTimePolicy	end_time	×	✓	×	×
Messaging::RelativeRequestTimeoutPolicy	relative_expiry	×	✓	×	×
Messaging::RelativeRoundtripTimeoutPolicy	relative_expiry	×	✓	×	×
PortableServer::LifespanPolicy	value	✓	✓	✓	×
PortableServer::IdUniquenessPolicy	value	✓	✓	×	×
PortableServer::IdAsssignmentPolicy	value	✓	✓	✓	×
PortableServer::POAManager	activate	✓	✓	✓	✓
PortableServer::POAManager	get_state	×	✓	×	×
PortableServer::POAManager	get_id	×	✓	✓	×
PortableServer::POA	create_poa	✓	✓	✓	×
PortableServer::POA	find_poa	✓	✓	✓	×
PortableServer::POA	destroy	✓	✓	✓	×
PortableServer::POA	create_lifespan_policy	✓	✓	✓	×
PortableServer::POA	create_id_uniqueness_policy	✓	✓	✓	×
PortableServer::POA	create_id_assignment_policy	✓	✓	✓	×
PortableServer::POA	the_name	✓	✓	✓	×
PortableServer::POA	the_parent	✓	✓	✓	×
PortableServer::POA	the_POAManager	✓	✓	✓	✓
PortableServer::POA	activate_object	✓	✓	✓	✓
PortableServer::POA	activate_object_with_id	✓	✓	✓	✓
PortableServer::POA	deactivate_object	✓	✓	✓	✓
PortableServer::POA	create_reference	✓	✓	✓	×
PortableServer::POA	create_reference_with_id	✓	✓	✓	×
PortableServer::POA	servant_to_id	✓	✓	✓	×
PortableServer::POA	servant_to reference	✓	✓	✓	×
PortableServer::POA	reference_to_servant	✓	✓	✓	×
PortableServer::POA	reference_to_id	✓	✓	✓	×
PortableServer::POA	id_to_servant	✓	✓	✓	×
PortableServer::POA	id_to_reference	✓	✓	✓	×
PortableServer::Current	get_POA	✓	✓	✓	×
PortableServer::Current	get_object_id	✓	✓	✓	×

Scope	Operation/ Feature	Minimum CORBA	CORBA/e Compact	SCA 4 Full Profile	SCA 4 Lightweight Profile
CORBA::Current	get_reference	×	✓	✓	×
CORBA::Current	get_servant	×	✓	✓	×
RTCORBA::ServerProtocolPolicy		×	×	✓	×
RTCORBA::PriorityModelPolicy	CLIENT_PROPAGATED	×	✓	✓	×
RTCORBA::PriorityModelPolicy	SERVER_DECLARED	×	✓	✓	×
RTCORBA::PriorityBandedConnectionPolicy	priority_bands	×	✓	✓	×
RTCORBA::Thread Pools	create_threadpool	×	✓	✓	×
RTCORBA::Thread Pools	create_threadpool_with_lanes	×	✓	✓	×
RTCORBA::Current	the_priority	×	✓	✓	×
RTCORBA::Mutex	lock	×	✓	×	×
RTCORBA::Mutex	unlock	×	✓	×	×
RTCORBA::Mutex	try_lock	×	✓	×	×
RTCORBA::RTORB	create_mutex	×	✓	×	×
RTCORBA::RTORB	destroy_mutex	×	✓	×	×
RTCORBA::RTORB	create_priority_model_policy	×	✓	✓	×
RTCORBA::RTORB	create_priority_banded_connection _policy	×	✓	✓	×
RTPortableServer::POA	activate_object_with_priority	×	✓	✓	×
CosNaming::NamingContext		o	✓	×	×
CosNaming::BindingIterator		o	✓	×	×
CosNaming::NamingContextExt		o	✓	×	×
CosEventComm::PushConsumer		o	✓	✓	×
CosEventComm::PushSupplier		o	✓	✓	×
CosEventComm::PullConsumer		o	✓	×	×
CosEventComm::PullSupplier		o	✓	×	×
CosEventChannelAdmin::ProxyPushConsumer		o	✓	×	×
CosEventChannelAdmin::ProxyPushSupplier		o	✓	×	×
CosEventChannelAdmin::ProxyPullConsumer		o	✓	×	×
CosEventChannelAdmin::ProxyPullSupplier		o	✓	×	×
CosEventChannelAdmin::ConsumerAdmin		o	✓	×	×
CosEventChannelAdmin::SupplierAdmin		o	✓	×	×
CosEventChannelAdmin::EventChannel		**o**	✓	×	×
CosLwLog::LogProducer		o	✓	✓	×
CosLwLog::LogConsumer		o	✓	✓	×
CosLwLog::LogStatus		o	✓	✓	×
CosLwLog::LogAdministrator		o	✓	✓	×
GIOP		✓	✓	✓	✓
IIOP		✓	✓	✓	✓

Table 4: Spectra ORB C++ Edition CORBA Profiles Mapping