

OpenFusion RTOrb Java™ Edition Version 1.5 Product Guide



OpenFusion RTOrb Java™ Edition

PRODUCT GUIDE



Part Number: RTJ-PG

Doc Issue 49, 15 April 2011

Copyright Notice

© 2011 PrismTech Limited. All rights reserved.

This document may be reproduced in whole but not in part.

The information contained in this document is subject to change without notice and is made available in good faith without liability on the part of PrismTech Limited or PrismTech Corporation.

All trademarks acknowledged.



CONTENTS

Table of Contents

Preface

About this Product Guide	xi
Contacts	xii

Introduction

OpenFusion RTOrb Java(tm) Edition	3
What is Real-time?	3
How RTOrb Provides for Real-time	4
Features, Standards and Compliance	4
Scope of this Guide for RTOrb	5

Installation and Configuration

Chapter 1	Installation	9
1.0.1	Conventions	9
1.1	Prerequisites	9
1.1.1	Operating Systems	10
1.1.2	System Variables	10
1.2	Installation Procedure	10
1.2.1	General	10
1.2.2	Preparation	11
1.2.3	Installation	11
1.2.3.1	Installing Using GUI Mode	11
1.2.3.2	Installing Using Command Line Mode	11
1.2.4	Install the Licence File	12
1.2.5	Testing the Installation	13
1.3	Uninstalling	13
Chapter 2	Configuration	15
2.1	Configuration Options and Properties	15
2.1.1	ORB Modes	17
2.1.1.1	Enterprise Mode	17
2.1.1.2	Soft Real-Time Mode	17
2.1.1.3	Hard Real-Time Mode	19
2.1.1.4	Multiple ORBs in a Single JVM	20
2.1.2	Configuration Properties	20
2.1.3	Threadpool Configuration	27
2.1.4	Messaging Configuration	27
2.1.5	Object Key Map	28

2.1.6 Logging	28
2.2 Configuring OpenFusion CORBA Services.....	29

Real-time Programming

Chapter 3	Reviewing CORBA Concepts	33
3.1	Basic Concepts	33
3.1.1	The ORB	33
3.1.1.1	Distributed Object Computing	33
3.1.1.2	Transparencies	34
3.1.2	Distributed Object Computing and CORBA.....	36
3.1.2.1	Interfaces	36
3.1.2.2	Programming with CORBA Interfaces.....	37
3.1.2.3	Delivering Requests Using an ORB	39
3.1.3	ORB Components	39
3.1.3.1	Abstraction	40
3.1.4	Terminology Explained.....	41
3.1.4.1	Clients and Servers	42
3.1.4.2	Object References.....	42
3.1.4.3	First Class Objects and Pseudo Objects	43
3.2	Portable Object Adapter	44
3.2.1	How the POA Works.....	45
3.2.1.1	POA Configuration.....	46
3.2.2	POA Policies	46
3.2.2.1	Standard POA Policies	46
3.2.2.2	POA Policy Summary	47
3.2.3	POA Manager	48
3.2.4	Object References, Keys, and IDs.....	48
3.2.5	Servants	48
3.2.6	Object Creation and Activation.....	48
3.2.7	Request Processing	49
Chapter 4	Introduction to Real-time Systems	51
4.1	Real-time Systems.....	51
4.1.1	Time- and Event-Triggered Systems.....	52
4.1.2	Developing Real-time Systems with RTOS	52
4.1.3	Predictability in Distributed Applications.....	54
4.1.4	Features and Non-Determinism.....	54
Chapter 5	Introduction to Real-time CORBA	57
5.1	Real-time Specification.....	57
5.1.1	Real-time CORBA Modules	57

5.1.2	Real-time ORB	58
5.1.3	Thread Scheduling	58
5.1.4	Real-time CORBA Priority	58
5.1.5	Native Priority and PriorityMappings	58
5.1.5.1	User-defined PriorityMappings	59
5.1.6	Real-time CORBA Current	59
5.1.7	Priority Models	59
5.1.8	Real-time CORBA Mutexes and Priority Inheritance	60
5.1.9	Threadpools	60
5.1.10	Priority Banded Connections	61
5.1.11	Non-Multiplexed Connections	61
5.1.12	Invocation Timeouts	61
5.1.13	Client and Server Protocol Configuration	61
5.1.14	Real-time CORBA Configuration	61
5.2	Real-time Portable Object Adapters	62
5.2.1	Priority Model	62
5.2.2	RTPOA	62
5.2.2.1	POA Activation Methods with Priority	62
5.2.3	Threads and Threadpools	62
5.2.3.1	Current	62
5.2.3.2	Threadpools	62
5.2.3.3	Thread Pool Operation Basic Mode	63
5.2.3.4	Laned Threadpool	64
5.2.3.5	Priority Banded Connections	64
5.2.4	RTPOA Current	64
5.2.5	Associations Between Pools and RTPOA	64
5.3	Priority Machinery	65
5.3.1	Priority Phenomena and Protocols	65
5.3.1.1	CORBA Priority	68
5.4	CORBA Mutex	69
5.4.1	Mutex Notifies in RT CORBA	69
5.4.2	Why Mutex Has a Priority Protocol	69
5.4.3	The Real-time CORBA Mutex Interface	70
Chapter 6	Introduction to Real-time Java	71
6.1	Real-time Extension to Java	71
6.1.1	Thread Scheduling and Dispatching	71
6.1.2	Memory Management	72
6.1.3	Synchronization	73
6.1.4	Asynchronous Event Handling	73
6.1.5	Asynchronous Transfer Of Control	73
6.1.6	Asynchronous Thread Termination	73

6.1.7 Physical Memory Access.	73
6.2 Further Reading and Examples Information	74

Programming with RTOrb

<i>Chapter 7</i>	Using the ORB	77
	7.1 Introduction	77
	7.1.1 Advice Notes	78
	7.1.2 Conventions.	79
	7.2 Using the IDL Compiler.	79
	7.3 Compiling Applications	81
	7.3.1 System and Environment Settings.	81
	7.3.2 Java Compiler	81
	7.3.2.1 Common Requirements	81
	7.3.2.2 Sun Java Real-Time System Requirements.	81
	7.3.2.3 IBM Websphere Real Time JVM.	82
	7.4 Deploying and Running Applications.	82
	7.4.1 RTOrb Run Scripts	82
	7.4.1.1 Sun Java Real-time System	83
	7.4.1.2 IBM Websphere Real Time JVM.	83
	7.4.2 Resolving Servers	84
	7.5 Application Creation Example	84
	7.6 Running OpenFusion CORBA Services.	87
<i>Chapter 8</i>	Creating Applications	89
	8.1 General	89
	8.2 A Simple Non Real-Time Application	89
	8.2.1 IDL Specification	90
	8.2.2 Java Implementation	91
	8.2.3 Server-side.	91
	8.2.4 Client-side	92
	8.3 A Simple Soft Real-time Application	94
	8.3.1 Server-side.	95
	8.3.2 Client-side	100
	8.4 A Simple Hard Real-time Application	102
	8.4.1 Server-side.	102
	8.4.2 Client-side	108
<i>Appendix A</i>	API Enhancements	115
	Classes and Methods	115
	Valuetypes and Factories.	116
	Examples	117

Bibliography	123
Index	127

Table of Contents

Preface

About this Product Guide

This *Product Guide* provides instructions and information needed to install, configure and use OpenFusion RTOrb Java(tm) Edition.

Intended Audience

The *Product Guide* is intended to be used by software developers who wish to use RTOrb to develop CORBA-based, real-time distributed applications in Java. RTOrb can also be used as a conventional, non real-time, high performance enterprise Java ORB for developers who do not need real-time capabilities.

Organisation

This *Product Guide* is divided into three major sections: *Installation and Configuration* which provides information on installing and configuring RTOrb; *Real-time Programming* provides background information on CORBA, Java and real-time programming; and *Programming with RTOrb* which describes how to create applications using RTOrb.

Conventions

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



Item of special significance or where caution needs to be taken.



Item contains helpful hint or special information.



Information applies to Windows (*e.g.* XP, 2003, Windows 7) only.



Information applies to Unix-based systems (*e.g.* Solaris) only.



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 xii, behave as hypertext links: jump to that section of the document 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. The Courier font also indicates file names.

Extended code fragments are shown as Courier font 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 Bold indicates user-related actions, such as **File > Save** from a menu.

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

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

Web:

Technical questions:

Sales enquiries:

<http://www.prismtech.com>

crc@prismtech.com (Customer Response Center)

sales@prismtech.com

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900

Fax: +44 (0)191 497 9901

A close-up, low-angle photograph of a computer keyboard, showing several keys in detail. The keys are white and set against a dark background. A white grid pattern is overlaid on the image, creating a sense of depth and perspective. The lighting is soft, highlighting the texture of the keys and the grid lines.

INTRODUCTION

OpenFusion RTOrb Java(tm) Edition

The OpenFusion RTOrb Java(tm) Edition brings together two powerful technologies: Real-time Java and Real-time CORBA. Real-time Java provides the power, flexibility and convenience of a platform independent, real-time language. Real-time CORBA provides the means of exploiting the benefits of Real-time Java within a distributed, platform independent real-time architecture.

The OpenFusion RTOrb Java(tm) Edition product (RTOrb, for short) combines the real-time Java language and real-time CORBA architecture into a single technology which can produce real-time, distributed applications which are deployable across diverse platforms with the minimum of effort.

What is Real-time?

There are several definitions available which state what *real-time* means, such as:

“Immediate, as an event is occurring.”¹

“The actual time during which physical events take place.”²

“The processing and visibility of transactions and information as they occur, and not on a periodic or batch basis.”³

“...computer systems that update information at the same rate as they receive data...”⁴

Current computer systems have physical restrictions which limit the ability to process information immediately, “*as an event is occurring*”- there are the inevitable processing speed and resource limits which affect how fast data can be processed. For the purpose of programming actual real-time applications, a more realistic definition of real-time has been adopted:

An application for which the requirements, design, or developers state that execution of application logic must or should occur within well-defined temporal conditions.⁵

Or in other words, the processing or completion of tasks is not instantaneous, but occurs within pre-defined *time limits*. This definition accepts the physical realities of our present computing machines and systems.

1. <http://www.hq.nasa.gov/office/pao/History/presrep95/r.htm>

2. http://www.telemet.com/weather_gloss_q_r.htm

3. <http://sun2.lenoir.cc.nc.us/~disted/distermc.htm>

4. *The American Heritage® Dictionary of the English Language*, Fourth Edition, © 2000 Houghton Mifflin Company.

5. This definition is as given in *Taking the Java™ Language into Uncharted Waters: Project Mackinac, Sun's RTSJ Implementation*, Bollella et. al., Sun Microsystems, Inc.

However, to complicate and possibly confuse matters, two different types of *real-time* have been identified, each relating to their ability to meet “well-defined temporal conditions”. The types are:

- *hard real-time* where the execution of the application logic must *always* meet the temporal requirements,
- *soft real-time* where the execution of the application logic may *sometimes* meet the temporal requirements.

A system where there are no well-defined temporal conditions is referred to as a *non real-time system*.

These definitions are important (even if they appear to complicate matters) since they provide flexibility as to the temporal stringency and capability which a system will be designed to achieve. Some systems must perform strictly within the temporal limits, whereas others can be more flexible, appreciating that it is likely to be more difficult and costly to create the more stringent systems.

How RTOrb Provides for Real-time

The language and architectural components of RTOrb address the practical issues of developing real-time applications for the real world, whether they need to meet the more demanding hard real-time requirements or the less demanding soft ones. Real-time Java and Real-time CORBA address the respective practical aspects of achieving hard or soft real-time requirements for distributed systems. Some aspects include:

- end-to-end predictable execution, thread scheduling and dispatching, along with the provision of distributable threads
- resource management, particularly memory management and allocation
- synchronisation, resource sharing and avoidance of *priority inversion*¹
- asynchronous event handling, transfer of control and thread termination
- interoperability and portability

Features, Standards and Compliance

The OpenFusion RTOrb Java(tm) Edition product complies with the following standards and specifications:

- OMG CORBA Specification, version 3
- GIOP Specification, version 1.3
- OMG Real-Time CORBA Specification, version 1.2

1. These aspects are to ensure that things happen in the correct sequence in order to meet specified temporal requirements.

- Real-Time Specification for Java, version 1.0

RTOrb complies with the following specific areas of the specifications:

- IDL Compiler (compliant with CORBA 2.3 specification)
- ORB Interface
- Value Type Semantics
- Abstract Interface Semantics
- DynAnys
- Interface Repository
- Portable Object Adapter
- Interoperability Overview
- ORB Interoperability Architecture
- Portable Interceptors
- CORBA Messaging (Messaging Quality of Service, Propagation of Messaging QoS)
- ETF

Please note, however, that RTOrb does not yet support:

- CORBA Component Model (component, event, home)
- policy domains
- message routing & ordering
- DCE ESIOP, Interworking, COM-CORBA or Automation mapping, Interoperability with non-CORBA systems
- fault tolerance or secure interoperability
- real-time scheduling
- Object Reference Templates
- DII/DSI
- AMI
- BiDirectional GIOP


Scope of this Guide for RTOrb

The goal of this guide is to help developers use RTOrb as quickly and effectively as possible. Its scope includes essential background information, in addition to installation, configuration and usage information.

It is beyond the scope of this manual to provide full coverage of RTOrb's underlying technologies, such as explaining real-time programming techniques and theory, or covering the Real-time Java or Real-time CORBA specifications. Information on these topics is available in the various documents listed in the bibliography.

This guide provides a technological overview which developers and architects can use as a starting point for understanding the intricacies of writing distributed, hard or soft real-time, CORBA-based, Java programs.

A number of useful, if not essential, references are provided in the *Bibliography*: readers are encouraged to use these references to develop understanding of this powerful technology.

A close-up, low-angle photograph of a computer keyboard, showing several keys in detail. The keys are white and the keyboard is set against a dark background. A white grid pattern is overlaid on the entire image, creating a technical or digital aesthetic. The text is centered in the upper half of the image.

INSTALLATION AND CONFIGURATION

CHAPTER

1 Installation

This chapter describes how to install OpenFusion RTOrb Java(tm) Edition (RTOrb). Please follow the procedures carefully.

The RTOrb installation files can be downloaded from the PrismTech Web site (www.prismtech.com).

1.0.1 Conventions

The following conventions are used in this chapter:

- Commonly used directories are shown as:
 - <OFJ_DIR> - where RTOrb is or will be installed
 - <JAVA_HOME> - root directory of the Java Virtual Machine (JVM) installation
- The directory paths and environment variable separator shown here use the UNIX forward-slash (/) and colon (:) separator conventions; Windows™ users should replace these separators with the standard DOS back-slash (\) and semi-colon (;) separators.
- Items which are unique to UNIX or Windows are shown using the *UNIX Only* or *Windows Only* icons, respectively. For example:

WIN

```
> SET CLASSPATH=.;%CLASSPATH%;
```

UNIX

```
% CLASSPATH=.:$CLASSPATH; export CLASSPATH
```

1.1 Prerequisites

RTOrb depends on underlying services and technologies. If these services and technologies are not properly installed and configured, then the OpenFusion RTOrb Java(tm) Edition cannot perform as intended. Accordingly, please check that your system meets each of the prerequisites described below before installing OpenFusion RTOrb Java(tm) Edition.

i

The currently supported platforms are listed on the RTOrb *Supported Platforms* web page. The *Supported Platforms* web page can be accessed from the *index.html* page located in the root directory where RTOrb is installed (<OFJ_DIR>).

Please refer to *Supported Platforms* and other *Release Notes* pages for the latest information about this distribution.

1.1.1 Operating Systems

For an up-to-date list of the operating systems that are supported by this release of OpenFusion RTOrb Java(tm) Edition please see [platforms.html](#) in the documentation directory.



If RTOrb is used as a real-time ORB,¹ then it **must** be installed on a real-time platform with a real-time JVM conforming to the Real-Time Specification for Java (RTSJ). The examples and settings given here show the IBM Websphere Real-Time JVM: if required, users should replace these settings with those used by their own real-time.

1.1.2 System Variables

The PATH and CLASSPATH environment variables must be set as described below.

The PATH must include:

- the directory where the Java interpreter is located (i.e. <JAVA_HOME>/bin)
- the directory where RTOrb's scripts are or will be located (<OFJ_DIR>/bin)

The CLASSPATH must include:

- the *current* directory reference, indicated by the full-stop or period character (".").



The ORB implementation code is in `lib/endorsed/ofj.jar`: it is not necessary to manually add `ofj.jar` to the classpath since the RTOrb scripts handle this.

Example Environment Variable Setting using the IBM Websphere JVM with RTOrb

Where:

`/opt/myOFJ` is the directory where RTOrb is or will be installed and

`/usr/local/j9rt` is the directory where the IBM WebSphere Real Time RT Linux is installed

UNIX

```
% PATH=/opt/myOFJ/bin:/usr/local/j9rt:$PATH
% export PATH
% CLASSPATH=.
% export CLASSPATH
```

1.2 Installation Procedure

1.2.1 General

All installed RTOrb files are placed in the RTOrb installation directory specified during installation - no files are stored in any of the UNIX system directories.

1. In other words, running RTOrb in its real-time mode (see *ORB Modes* on page 17.)

RTOrb is installed using a Java-based *Setup* program. This program can be run using a Graphical User Interface (GUI Mode) or from the command line (Command Line Mode) which enables the installation to be run from a script.

1.2.2 Preparation

It is recommended that any existing RTOrb installation be removed before installing the current version (see *Uninstalling* on page 13). Please note the following warning.



Uninstalling OpenFusion RTOrb Java(tm) Edition removes all RTOrb files, including the executables, licence, configuration, and data files located in the RTOrb sub-directories. If these files are required, then they should be backed-up prior to uninstalling.

1.2.3 Installation

As described above, RTOrb can be installed using an interactive graphical user interface (GUI Mode) or using commands entered on the command line (Command Line Mode). Using GUI Mode is generally the more popular method, however the Command Line Mode is useful when automating the installation with a script.

1.2.3.1 Installing Using GUI Mode

Step 1: Run `java Setup` (*without* any options) from the command line, as follows:

Change to the directory containing RTOrb's `Setup.class` or `Setup.jar`. `Setup` can be run directly on the command line or from a script:

```
% java Setup
```

This will display the RTOrb installation's graphical user interface.

Step 2: Follow the instructions displayed in the GUI, selecting the services and components you want to install.

1.2.3.2 Installing Using Command Line Mode

Step 1: Run `java Setup` from the directory selected above *with* the options shown below.

Change to the directory containing the RTOrb `Setup.class` or `Setup.jar`. `Setup` can be run directly on the command line or from a script.

- For JDK versions *prior* to JDK version 1.6, use:

```
% java Setup <-list | [<OFJ_DIR> [components]]>
```

- For JDK version 1.6 or later, use:

```
% java -jar Setup.jar <-list | [<OFJ_DIR> [components]]>
```

where

-list will list all available services and components, without performing the installation

<OFJ_DIR> is the directory where RTOrb is to be installed

[components] is the list of components and services to be installed; if no components are specified, then all components will be installed

Example 1

List all available services and components

```
% java Setup -list
```

Example 2

Install all services and components to /opt/myOFJ.

```
% java Setup /opt/myOFJ
```

Example 3

Install the Naming Service to /opt/myOFJ.

```
% java Setup /opt/myOFJ Naming
```

1.2.4 Install the Licence File

A valid RTOrb license file must be placed into the *<OFJ_DIR>/etc* directory after RTOrb has been installed. Please note that OpenFusion RTOrb Java(tm) Edition will not run without a valid licence file.

Licence files are provided by PrismTech for services or products which have been purchased. Contact PrismTech for purchasing details (see *Contacts* on page xii).

Evaluation licences are provided when an OpenFusion product is downloaded from the PrismTech Web site (at www.prismtech.com.)

1.2.5 Testing the Installation

The RTOrb installation can be tested by checking:

- the RTOrb is running properly by running an example (refer to `<OFJ_DIR>/examples/index.html`). The examples can be run from any directory since the precompiled RTOrb examples are in the `ofj.jar` file (which is placed in the classpath by the RTOrb *run* scripts).
- For example, using the non real-time Hello example, run the Server and Client programs in separate windows.

```
% run com.prismtech.ofj.examples.corba.hello.Server
% run com.prismtech.ofj.examples.corba.hello.Client
```

If both Server and Client run successfully, the Server window should display:

```
GreetingServer running... awaiting calls
GreetingService called by Client hello Client
```

1.3 Uninstalling

This section describes the procedure for uninstalling the OpenFusion RTOrb Java(tm) Edition.



Uninstalling RTOrb removes all RTOrb files, including the executables, licence, configuration, and data files located in the RTOrb sub-directories. If these files are required, then they should be backed-up prior to uninstalling.

Step 1: Stop any running OpenFusion services.

Step 2: Backup any data, licence or other required files which are in the OpenFusion directories.

Step 3: Run the *uninstall* utility (located in the *bin* directory):

```
% <OFJ_DIR>/bin/uninstall
```



Windows users can also use **Start | Programs | OpenFusion | Uninstall OpenFusion** to start the utility.

The utility displays a confirmation dialogue box which asks if you wish to proceed with uninstallation. Clicking the **Yes** button will uninstall RTOrb.

2 Configuration

2.1 Configuration Options and Properties

Although RTOrb's internal settings are pre-configured with standard or typically used values (which enables RTOrb to be installed as quickly and easily as possible), these settings can be given user-defined, custom values to meet specific operational and application requirements.

A user's custom properties file can be loaded into RTOrb by passing the properties file's pathname to RTOrb's *custom.props* property: in other words, the value of *custom.props* is the path to the properties file that contains a list of the properties that should be loaded by RTOrb.

Being able to load a custom properties file can be useful for application-specific settings which need to be distributed with the code.

RTOrb's user-configurable options and properties can be set:

- using the command line
 - allows system properties to be assigned by passing the `-D<property>=<value>` switch to the program. When using the supplied *run* or *runrt* scripts, this should be the first argument.
- programmatically by passing *Properties* objects to *ORB.init()* using the form:

```
Properties props =
    new Properties(System.getProperties());
props.setProperty ("PropertyKey", "PropertyValue");
org.omg.CORBA.ORB orb = ORB.init(args, props);
```

- passing values to RTOrb as program arguments. The arguments that are accepted in this way are *ORBid*, *ORBInitRef*, *ORBDefaultInitRef*, *ORBListenEndpoints* and *ORBpriorityrange*. Note that *ORBpriorityrange* is not described in the CORBA 3 specification. Also note that:
 - *ORBid <name>* overrides the default ORB ID to set the name of this ORB. The default ORB ID uses the form *OFJ<n>*, where *<n>* is an integer which starts at 0 and is incremented for each default ID.

- *ORBListenEndpoints* sets the interface for the RootPOA to listen on. The format for *ORBListenEndpoints* *<argval>* is

```
<argval> := <hostname>:<portspec> | <portspec>
<portspec> := <portnum> | <portnum>,<portnum>
<hostname> := STRING | IPADDRESS
<portnum> := INTEGER
```

- *ORBListenEndpoints* overrides the *ofj.etf.default-port_low*, *ofj.etf.default-port_high* and *ofj.etf.default-host* properties. For instance, if *-ORBListenEndpoint 1234* has been specified this will override the port properties (if set).

If *ofj.listen-on-all-interfaces* is *true* (the default), the ORB will listen on all interfaces on port 1234.

If *listen-on-all-interfaces* is *off* the ORB will listen on the default interface on port 1234.

If *-ORBListenEndpoint 192.168.100.100:1234* has been set this will force the ORB to only listen on the above interface and port.

A child POA will use the RootPOA's listeners unless it has been created with a Server Protocol Policy. This policy will contain one or more Protocols, each of which will specify an interface on which the child POA will listen. Child POAs do not inherit listeners from parent POAs.

For example:

```
ORBListenEndpoints 12000 to listen on port 12000
```

```
ORBListenEndpoints 12000, 14000 on ports 12000 through 14000, inclusive
```

- *ORBpriorityrange* allows the user to specify a minimum and maximum priority to use in the real-time ORB.

Examples of the *run* and *runrt* commands are:

```
% run com.prismtech.ofj.examples.corba.hello.Server -ORBId ofj27
```

```
% runrt com.prismtech.ofj.examples.corba.hello.Server -ORBId ofj27
```

- using RTOrb's configuration file
 - the configuration file, *ofj.properties*, is located in *<OFJ_DIR>/classes*
 - assign property values using the form of *prop=value* (without spaces)



Properties are set in order: properties loaded later will override properties loaded earlier. In addition to the different methods of loading properties files (as described above), there are also several different properties files located in different directories. The properties files are loaded into RTOrb in the following order:

- configuration file in *java.home/lib*
- configuration file in *user.home*
- configuration file on the classpath
- custom properties file named in *custom.props*
- command line properties
- programmatically by passing *Properties* objects to *ORB.init()*

2.1.1 ORB Modes

OpenFusion RTOrb Java(tm) Edition supports one non-real-time mode and two real-time modes of operation. Each of these modes offers different levels of determinism and ease of use. The modes are:

- **Enterprise Mode** - In this mode, RTOrb functions as a conventional, non-real-time ORB.
- **Soft Real-Time Mode** - This mode offers soft real-time determinism combined with ease of use comparable with conventional programming in the Java environment.
- **Hard Real-Time Mode** - This mode offers very high levels of determinism, but is the most complex to program.

2.1.1.1 Enterprise Mode

Enterprise mode has been designed to enable RTOrb to be used as a replacement for non-real-time ORBs enabling a single ORB to meet the full range of project requirements.

Set the following properties as shown to run RTOrb as a non real-time enterprise ORB:

```
org.omg.CORBA.ORBClass=com.prismsotech.ofj.orb.ORB
org.omg.CORBA.ORBSingletonClass=com.prismsotech.ofj.orb.ORBSingleton
```



Note that Enterprise Mode is the default which is set by the RTOrb *run* script (see *RTOrb Run Scripts* on page 82).

2.1.1.2 Soft Real-Time Mode

The soft real-time mode has been designed to meet the needs of users who are developing systems that have soft real-time characteristics. When used in conjunction with a JVM with a real-time garbage collector, performance characteristics suitable for soft real-time systems can be achieved whilst retaining

the memory management techniques of conventional Java applications. The soft real-time mode is therefore significantly easier to use than the other real-time modes supported by OpenFusion RTOrb Java(tm) Edition.

2.1.1.2.1 Memory Management

The soft real-time mode is also called the "Full Heap mode". All internal objects and all objects returned to the application are conventionally allocated in heap memory. All internal threads are `javax.realtime.RealtimeThreads` and run in heap memory.

The methods of the ORB, of the POAs and of the CORBA objects have been designed to be called in any of the memory types supported by the RTSJ (Immortal Memory, Scoped Memory), but the soft real-time mode has been designed to be called only in heap memory. If your application carries out memory management and runs in Immortal or Scoped Memory, you should use hard real-time mode.

The methods of the ORB, of the POAs and of the CORBA objects can be called in a standard `java.lang.Thread` or in a `javax.realtime.RealtimeThread`. Since the soft real-time mode uses only heap memory, it cannot be used with `NoHeapRealtimeThreads`.

The soft real-time mode implements the RTCORBA API but it can also be used with no code changes by a legacy application designed for an enterprise ORB.

2.1.1.2.2 RTCORBA API Restrictions

The soft real-time mode implements the RTCORBA API.

When `the_priority(short)` method is called on a non real-time Thread (for example, within Soft RealTime mode) and if the priority value exceeds the range of the `java.lang.Thread`, then the priority value will be set to `Thread.MAX_PRIORITY`.

2.1.1.2.3 Advantages and Disadvantages of the Soft Real-Time Mode

The major advantage of the soft real-time mode is that, as everything is allocated in heap and can be garbage collected, the application is not required to carry out any explicit memory management. This greatly simplifies the application.

Secondly, this mode allows the porting of a legacy application to a real-time application very easily and with no code changes.

The main disadvantage of this mode is its lack of strong determinism. Whilst the ORB avoids the use of the garbage collector as much as possible, the garbage collector can still pre-empt your application, potentially causing jitter in the response times.

2.1.1.2.4 Configuration of the Soft Real-Time Mode

When invoking RTOrb add the following properties to the command line:


```
-Dorg.omg.CORBA.ORBClass=com.prismtech.ofj.rtorb.RTORB
-Dorg.omg.CORBA.ORBSingletonClass=com.prismtech.ofj.orb.ORBSingleton
-Dofj.rtmode=soft
```



Note that the `ofj.rtmode` property is in the configuration file `ofj.properties`. It defaults (for real-time mode) to `soft`. Also, the `ofj.rtmode` property can be set in application code like other properties, but it must be set before any calls are made to the ORB code.

2.1.1.3 Hard Real-Time Mode

The hard real-time mode has been designed to meet the needs of users who are developing systems that have hard real-time characteristics. Using this mode, it is possible to create applications that exhibit low levels of jitter (less than 1 ms). However, in order to achieve this, applications must use features of RTSJ such as Scoped Memory and No Heap Real Time Threads. In order to avoid interference from the garbage collector, applications must not allocate any object in heap memory and must, as a consequence of this, implement their own memory management schemes. The hard real-time mode is therefore more complex to use than the other modes supported by RTOrb Java Edition.

2.1.1.3.1 Memory Management

The hard real-time mode is designed to exhibit strong determinism and to only use `javax.realtime.NoHeapRealtimeThreads`. Consequently, this mode is designed to use only Immortal Memory and Scoped Memory. All internal objects and objects returned to the application will be created in either Immortal Memory or in Scoped Memory. All ORB and POAs initialisation methods have to be run in Immortal Memory and in general it is recommended that any CORBA methods are run in Immortal or Scoped memory.

In this mode, the application can use `javax.realtime.RealtimeThreads` or `javax.realtime.NoHeapRealtimeThreads` but the ORB cannot be used in a standard `java.lang.Thread`.

2.1.1.3.2 Advantages and Disadvantages of the Hard Real-Time Mode

The advantage of the hard real-time mode is the high level of determinism that can be achieved. The cost of this is increased application complexity since applications must implement their own memory management.

2.1.1.3.3 Configuration of the Hard Real-Time Mode

When invoking RTOrb add the following properties to the command line:

```
-Dorg.omg.CORBA.ORBClass=com.prismtech.ofj.rtorb.RTORB
-Dorg.omg.CORBA.ORBSingletonClass=com.prismtech.ofj.orb.ORBSingleton
-Dofj.rtmode=hard
```

i

Note that the `ofj.rtmode` property is in the configuration file `ofj.properties`. Also, the `ofj.rtmode` property can be set in application code like other properties, but it must be set before any calls are made to the ORB code.

2.1.1.4 Multiple ORBs in a Single JVM

RTOrb supports multiple ORBs of different types, in other words, a combination of enterprise, soft real-time and hard real-time in a single JVM.

In order to create different ORBs within the same JVM inline within the code, use a combination of the `org.omg.CORBA.ORBClass` and `ofj.rtmode` properties and pass these arguments to `ORB.init ()`.

Examples

Enterprise mode

```
Properties props = new Properties ();
props.put ("org.omg.CORBA.ORBClass",
           "com.prismtech.ofj.orb.ORB");
props.put ("org.omg.CORBA.ORBSingletonClass",
           "com.prismtech.ofj.orb.ORBSingleton");
```

Soft real-time mode

```
Properties props = new Properties ();
props.put ("org.omg.CORBA.ORBClass",
           "com.prismtech.ofj.rtorb.RTORB");
props.put ("org.omg.CORBA.ORBSingletonClass",
           "com.prismtech.ofj.orb.ORBSingleton");
props.put ("ofj.rtmode", "soft");
```

Hard real-time mode

```
Properties props = new Properties ();
props.put ("org.omg.CORBA.ORBClass",
           "com.prismtech.ofj.rtorb.RTORB");
props.put ("org.omg.CORBA.ORBSingletonClass",
           "com.prismtech.ofj.orb.ORBSingleton");
props.put ("ofj.rtmode", "hard");
```

2.1.2 Configuration Properties

The tables shown in this section describe the properties that can be used to configure RTOrb. Note that the properties for Threadpools and Messaging are given under Sections 2.1.3, *Threadpool Configuration*, on page 27 and 2.1.4, *Messaging Configuration*, on page 27, respectively

Table 1 General Settings

Property	Description	Default
ORBInitRef.<name>=<ref>	Defines an initial reference. The <name> is the name of the service, the <ref> is the object reference. For example: ORBInitRef.MyService=file:///home/me/MyRef.ior	None
OF.License.File	The directory where the RTOrb license file is installed on your system. This value is automatically set by the RTOrb run and runrt scripts. However, if the license file is placed in another location, this value must be changed accordingly.	<OFJ_DIR>/etc
ofj.log=[OFF FATAL ERROR WARN INFO DEBUG TRACE ALL]	The different log levels at which RTOrb can log events. (See also section 2.1.6, <i>Logging</i> , on page 28.)	WARN
ofj.log.timestamp=[off on]	If logging is on, this property determines whether time stamps are included in the log.	off
ofj.log.timestamp-format=[ms time date-time]	If logging is on and ofj.log.timestamp is set to <i>on</i> , this property determines the style or format of time stamps. (See also section 2.1.6, <i>Logging</i> , on page 28.)	ms
ofj.log.location=[off on]	If logging is on, this property determines whether the file and line number are included in the log.	off
ofj.log.thread=[off on]	If logging is on, this property determines whether the current thread identifier is included in the log.	off

Table 2 ORB Configuration

Property	Description	Default
ofj.rtmode=[soft hard]	The real-time mode of the ORB.	soft
ofj.orb.memory	The size of the temporary memory area used to read replies on client side. Also the size of the several temporary memory areas used to call interceptors on client side.	200000

Table 2 ORB Configuration

Property	Description	Default
<code>ofj.orb.objectKeyMap.<name></code>	Allows more readable corbaloc URLs by mapping the actual object key to an arbitrary string. See Section 2.1.5, <i>Object Key Map</i> , on page 28.	no default value
<code>ofj.orb_initializer.fail_on_error=[on off]</code>	Controls whether ORB initialisation is allowed to continue or not if ORBInitializers fail.	off
<code>ofj.giop_minor_version</code>	The GIOP minor version number to be used for newly created IORs	2
<code>ofj.native_codeset_for_chars</code>	The code set that will be declared as the native code set for <i>chars</i> in newly created IORs.	ISO8859_1
<code>ofj.native_codeset_for_wchars</code>	The code set that will be declared as native code set for <i>wchars</i> in newly created IORs.	UTF16
<code>ofj.send-tag-orb-type</code>	Flag to indicate whether the TAG_ORB_TYPE TaggedComponent should be included in IORs	false

Table 3 objectcache Properties

Property	Description	Default
<code>ofj.objectcache.cachedObjectClasses</code>	A comma-delimited list of class names to be managed by object. A list of names must be supplied if object caching is required.	None.
<code>ofj.objectcache.<classname>.cacheClass</code>	The class name of the object cache implementation. The default is: <code>com.prismtech.ofj.util.objectcache.ObjectCacheImplementation</code>	See <i>Description</i> .
<code>ofj.objectcache.<classname>.initialSize</code>	The initial number of objects an object cache contains.	0
<code>ofj.objectcache.<classname>.growByAmount</code>	The number of objects allocated at the same time if the cache is empty but the <code>maximumSize</code> has not been reached.	1

Table 3 objectcache Properties (Continued)

Property	Description	Default
<code>ofj.objectcache.<classname>.maximumSize</code>	<p>The maximum number of objects allocated by the cache.</p> <p>The value of <code><classname></code> should be set to one of:</p> <ul style="list-style-type: none"> • a specific class to which the property should be applied, for example: <pre>ofj.objectcache.mypackage. MyStruct.initialSize=0</pre> • default - to define a property value for all object cache instances, for example: <pre>ofj.objectcache.default-initialSize =0</pre> 	128

Table 4 POA Configuration Properties

Property	Description	Default
<code>ofj.extra.agents</code>	<p>Agents are threads that handle requests and run servant code on the server side. These are only used by Enterprise POAs and there is a single agent threadpool per ORB instance. This pool is used by all POAs within an ORB.</p> <p><code>extraAgents</code> is the number of extra agents that can be run to handle simultaneous requests. These extra Agents are added to the pool on <code>rootPOA</code> creation. By default, a new Agent is created on demand for each new connection.</p> <p>If all available agents are busy then no data is read from the connection(s).</p> <p>There is no explicit buffering of messages.</p> <p>The agent threads are only stopped when the <code>RootPOA</code> is destroyed. Therefore if some connections are closed the number of agent threads will not decrease.</p> <p>The number of threads is the number of connections plus the number of extra agents plus the number of POAs. For example, if there are 3 clients, and <code>ofj.extra.agents</code> has the value 5, and there is 1 POA, then there can be up to 9 agent threads. If there are 4 clients and <code>ofj.extra.agents</code> has the value 2, and there are 2 POAs, then there can be up to 8 threads.</p>	5
<code>ofj.rtpoa.memory</code>	The amount of memory used by the real-time POA Reader on the server side.	200000

Table 5 ETF Configuration Properties

Property	Description	Default
<code>ofj.etf.use-default-transport</code>	When set to <code>true</code> , the default transport plugin, <code>IIOP</code> , are used.	<code>true</code>
<i>The following properties are TCP protocol properties.</i>		

Table 5 ETF Configuration Properties (Continued)

Property	Description	Default
<code>ofj.etf.default-send-buffer-size</code>	Provides a hint to the underlying platform about the size of the buffers used for sending data via a socket. The platform's default value should be used when this property is set to <code>0</code> .	0
<code>ofj.etf.default-recv-buffer-size</code>	Provides a hint to the underlying platform about the size of the buffers used when receiving data via a socket. The platform's default value should be used when this property is set to <code>0</code> .	0
<code>ofj.etf.default-keep-alive</code>	Send keep alive messages on a connection. The time between keep alives is platform-dependant, but is typically 2 hours. The configurability of this property is also platform-dependant.	false
<code>ofj.etf.default-dont-route</code>	Specifies that data should not be sent via a gateway, but sent directly to connected hosts. This property is currently ignored since it is not possible to set this option on Java sockets.	false
<code>ofj.etf.default-no-delay</code>	Disables <i>Nagle's Algorithm</i> when set to <code>false</code> . This algorithm can reduce network congestion when many very small messages are sent, but in some circumstances it can interact badly with TCP delayed ACKs. The algorithm is probably not helpful for most CORBA applications.	false
<code>ofj.etf.default-port_low</code>	This sets the default value of the lower bound (lowest port address) of the range of TCP ports used by POAs. If the lowest and highest port range values are not set or are set to negative values, then RTOrb will select an ephemeral port to listen on. If lowest and highest port range values are set greater than 0, then RTOrb will select an available port within that range.	0

Table 5 ETF Configuration Properties (Continued)

Property	Description	Default
ofj.etf.default-port_high	<p>This sets the default value of the upper bound (highest port address) of the range of TCP ports used by POAs.</p> <p>See <i>ofj.etf.default-port_low</i> above for details</p>	0
ofj.etf.default-host	<p>The default hostname used by POAs. The name chosen by the ORB can be found by running the <i>com.prismtch.ofj.util.NetworkInfo</i> utility.</p> <p>This property accepts <i>hostname</i> as a text name or as an IP address, either of which will be written into the IOR.</p>	
ofj.etf.listen-on-all-interfaces	<p>By default, the ORB now listens on all network interfaces, and chooses a non-loopback interface to put in IOR profiles.</p> <p>If <i>listen-on-all-interfaces</i> is <i>false</i>, the ORB will choose to listen on a non-loopback interface if available, or the loopback interface if not. The chosen address will be used in IOR profiles.</p> <p>If <i>listen-on-all-interfaces</i> is <i>false</i>, it will listen on the interface named by the <i>default-host</i> property if set, or the first non-loopback interface if <i>default-host</i> is not set.</p>	true
ofj.etf.use-names	<p>If <i>use-names</i> is <i>true</i>, the ORB will attempt to use a textual hostname rather than a numeric IP address in IOR profiles.</p>	false
ofj.etf.server.reuseAddress	<p>Enables applications to bind to a TCP socket even if a previous connection to the socket is in a time out state. When enabled, this property can be used to allow applications, typically using a well known socket address, to bind to the socket without needing to wait for the time out period to expire.</p>	false

2.1.3 Threadpool Configuration

If a threadpool is not created in the application, then a default threadpool with one lane that allows request buffering is created and the default values in the table above are assigned programmatically. If different values are required then the properties can be set to override the default values. See *Table 6* for the configuration properties.

Table 6 Threadpool Configuration

Property	Description	Default
<code>ofj.rtorb.deftp.stacksize</code>	The stack size, in bytes, that each thread in the threadpool must have allocated.	65535
<code>ofj.rtorb.deftp.staticthreads</code>	The number of threads that are pre-created and assigned to that threadpool at the time of the threadpool creation.	5
<code>ofj.rtorb.deftp.dynamicthreads</code>	The number of additional threads that may be created dynamically when the static threads are all in use and an additional thread is required to service an invocation.	0
<code>ofj.rtorb.deftp.priority</code>	The CORBA priority of the only lane in the default threadpool.	16384
<code>ofj.rtorb.deftp.maxbuffreqs</code>	The maximum number of requests that will be buffered by this threadpool if all available static and dynamic threads are in use and the capability to borrow threads from lower priority lanes is exhausted.	328
<code>ofj.rtorb.deftp.maxreqsbuffsize</code>	The maximum amount of memory, in bytes, that the buffered requests may use.	65535
<code>ofj.hardrt.minpriority</code>	The minimum priority assigned to threads running in hard real-time mode.	20

2.1.4 Messaging Configuration

Table 7 Messaging Configuration Properties

Property	Description	Default
<code>ofj.messaging.syncnone.threads</code>	The number of threads in the threadpool used for sending oneway <code>sync_none</code> requests.	5

Table 7 Messaging Configuration Properties (Continued)

Property	Description	Default
<code>ofj.messaging.syncnone.priority</code>	The priority of the threads used for sending oneway <code>sync_none</code> requests.	16384

JVM Configuration

When running in hard real-time mode, the JVM should be configured to allow the ORB to:

- allocate all its data structures in *immortal memory*
- provide enough scoped memory to allocate threadpool scopes.

The scopes should be dimensioned to allow the ORB to allocate at least all data structures exchanged through the IDL.

The RTOrb `runrt` script provided with OpenFusion RTOrb Java(tm) Edition sets the default size for immortal and scoped memory in the JVM.



This memory size should be increased for applications which send large messages.

2.1.5 Object Key Map

An *objectKeyMap* facility exists that allows references to transient objects since it is not possible to construct a transient object with a readable key. This functionality may also be used with persistent objects. This `objectKeyMap` property allows more readable CORBALOC URLs by mapping the actual object key to an arbitrary string. The mapping currently allows clients of a service to access it using either IOR or file URL references. See Table 2, *ORB Configuration*, on page 21.

The `ofj.orb.objectKeyMap.<name>` is configured using the `ofj.properties` file and by using the proprietary `com.prismtech.ofj.orb.ORB.addObjectKey(String name, String value)` function. For example:

```
ofj.orb.objectKeyMap.MyService=file:///home/me/MyService.ior
```

A client wishing to use a CORBALOC reference to a server may then use:

```
-DORBInitRef.MyService=corbaloc:iiop:<host>:<port>/MyService
```

2.1.6 Logging

The `ofj.log` property values definitions are:

OFF - is the highest logging level and turns logging off

FATAL - very severe error events that can possibly cause the application to abort

ERROR - error events that might allow the application to continue running

WARN - potentially harmful situations

INFO - coarse grained information showing the application's progress

DEBUG - fine grained information that can be useful for debugging an application

TRACE - very finely grained information, even more so than the *DEBUG* level

ALL - the lowest logging level and turns all logging on

Logs can optionally include timestamps, file location, and current thread. These are controlled by the following properties:

ofj.log.timestamp

on - include timestamps

off - no timestamps

Defaults to *off*.

ofj.log.timestamp-format

ms - a time of the form $t=<time>$ where $<time>$ is UTC in milliseconds since 1970

time - a time in ISO 8601 format, e.g. 14:12:42.039. The time is in the local time zone.

date-time - an ISO 8601 date followed by a time,

e.g. 2010-03-03 11:23:02.324

Defaults to *ms*.

ofj.log.location

on - include location, i.e. file and line number

off - no location

Defaults to *off*.

ofj.log.thread

on - include the current thread identifier

off - no thread

Defaults to *off*.

2.2 Configuring OpenFusion CORBA Services

PrismTech's OpenFusion CORBA Services can be used with RTOrb. RTOrb includes the OpenFusion Naming Service. Refer to the *OpenFusion CORBA Services Naming Service Guide* and the *OpenFusion CORBA Services System Guide* (especially the *Common Configuration Properties* section).

If you are using other OpenFusion services with RTOrb, then refer to that service's guide as well as the *System Guide*.

A close-up, low-angle view of a computer keyboard, showing several keys in detail. The keys are white and set against a dark background. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The text "REAL-TIME PROGRAMMING" is centered in the upper half of the image.

REAL-TIME PROGRAMMING

3 Reviewing CORBA Concepts

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 Basic Concepts

3.1.1 The ORB

A core element of CORBA is the *Object Request Broker*, 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. ORBs are one of the principal enabling technologies in the field of distributed object computing.

3.1.1.1 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.

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

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.

3.1.1.2 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.

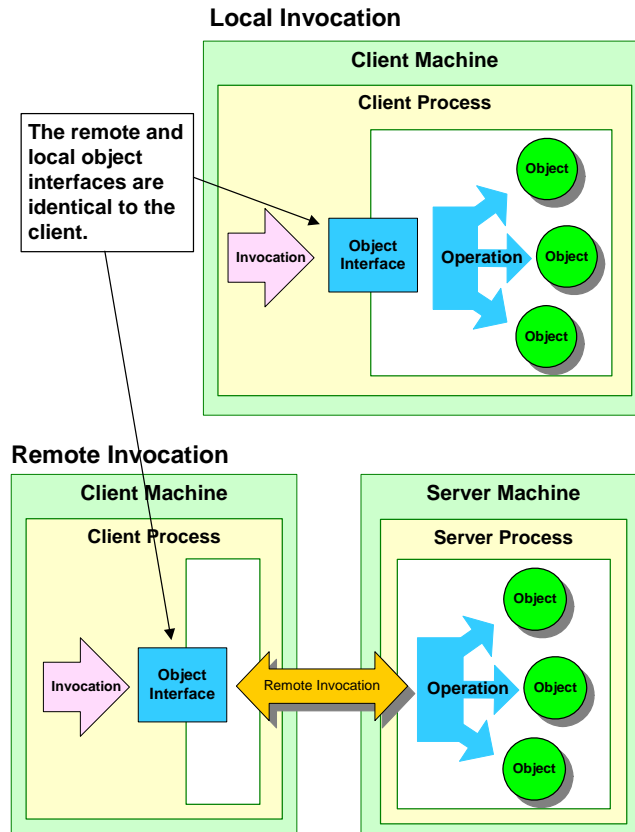


Figure 1 Remote Invocations and Location Transparency

The ORB provides the location transparency in the CORBA model. 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.

3.1.2 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.

3.1.2.1 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.

3.1.2.2 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.

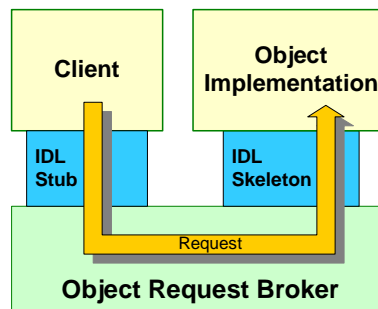


Figure 2 ORB Component Relationships

3.1.2.2.1 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.

3.1.2.2.2 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.

3.1.2.2.3 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.

3.1.2.3 Delivering Requests Using an 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 OpenFusion RTOrb Java(tm) Edition implementation.

3.1.2.3.1 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.

3.1.3 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.

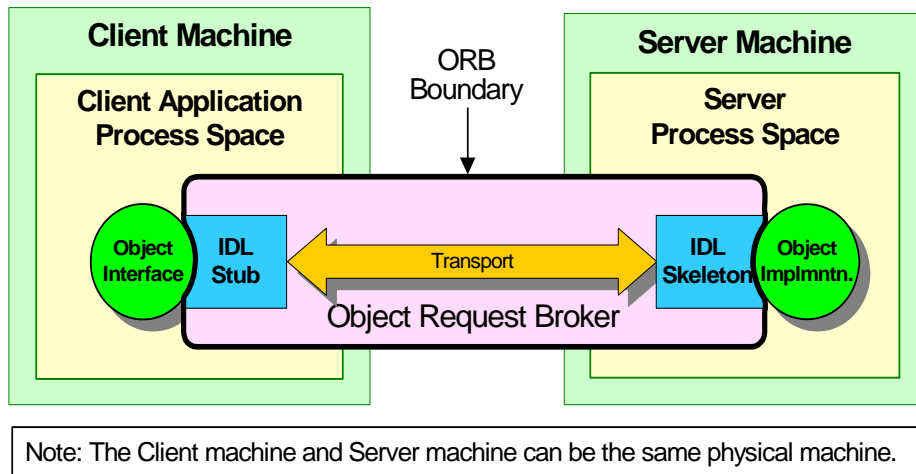


Figure 3 The ORB as an Abstraction

3.1.3.1 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.

3.1.4 Terminology Explained

Figure 4 is an adaptation from the CORBA 2.3 specification. 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.

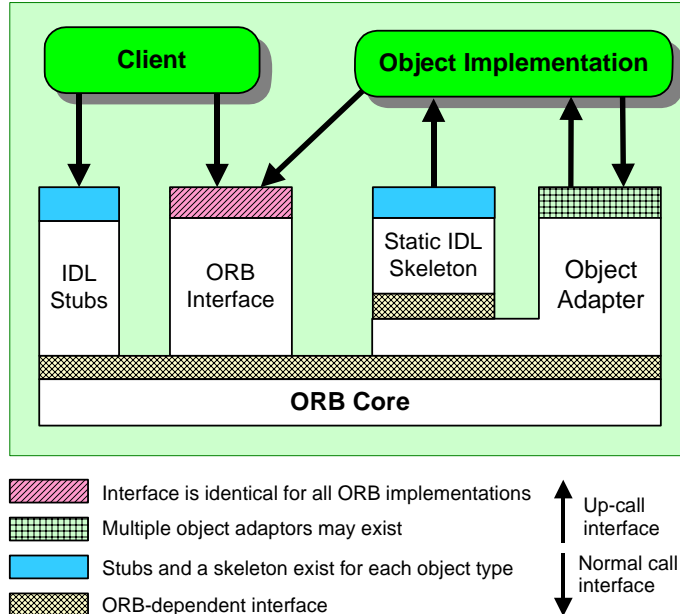


Figure 4 The Structure of Object Request Broker Interfaces

3.1.4.1 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.

3.1.4.2 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.

3.1.4.3 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

A first class object may also be referred to as a *righteous* object.

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.

3.1.4.3.1 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.

3.1.4.3.2 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)

3.2 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
- supports object implementations that inherit from static skeleton classes, as well as Dynamic Skeleton Interface (DSI) implementations

3.2.1 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.

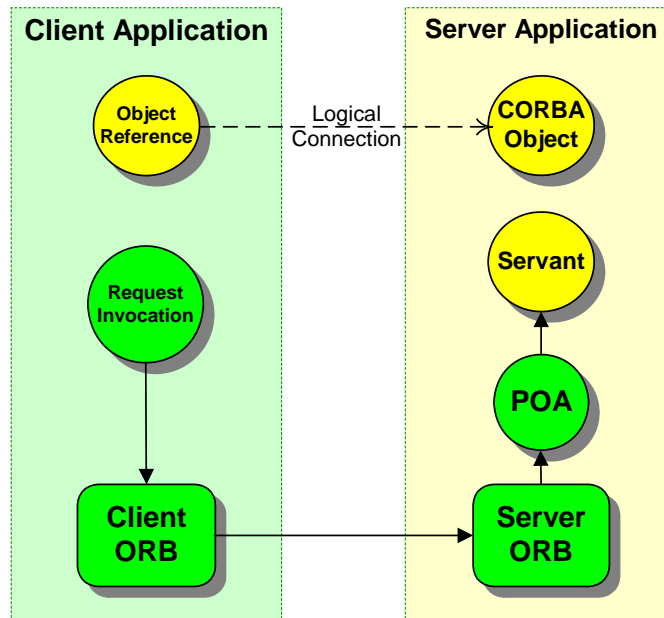


Figure 5 Request Dispatching

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.

3.2.1.1 POA Configuration

The RootPOA is configured *via* OFJ properties or arguments. If the `listen-on-all-interfaces` property is set (the default case), it will listen on all available interfaces. If not, it will listen on the interface named by the `default-host` property if set, or the first non-loopback interface if `default-host` is not set.

Alternatively, one or more `-ORBListenEndpoints <argval>` arguments can be given. Each argument specifies one interface to listen on. The format of `<argval>` is:

```
<argval> := <hostname>:<portspec> | <portspec>
<portspec> := <portnum> | <portnum>,<portnum>
<hostname> := STRING | IPADDRESS
<portnum> := INTEGER
```

A child POA will use the RootPOA's listeners unless it has been created with a Server Protocol Policy. This policy will contain one or more Protocols, each of which will specify an interface on which the child POA will listen. Child POAs do not inherit listeners from parent POAs.

3.2.2 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` to the `createPOA` method.

3.2.2.1 Standard POA Policies

3.2.2.1.1 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.

3.2.2.1.2 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`.

3.2.2.1.3 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`.

3.2.2.2 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.

3.2.3 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:

Table 8 POA States

POA State	Meaning
ACTIVE	Calling <code>activate</code> on the POAManager allows requests to flow to the POAs that it controls.
INACTIVE	This state is entered when POAs are to be shut down using the <code>deactivate</code> operation
DISCARDING	In this state all incoming requests (whose processing has not yet begun) will be discarded. The POA enters this state through the <code>discard_requests</code> operation when in the active or holding state.
HOLDING	In this state all incoming requests (whose processing has not yet begun) will be discarded. The POA enters this state through the <code>discard_requests</code> operation when in the active or holding state.

3.2.4 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`.

3.2.5 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.

3.2.6 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

3.2.7 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 Introduction to Real-time Systems

This chapter expands on the short introduction given earlier and introduces some of the essential aspects of real-time systems programming.

4.1 Real-time Systems

The term *real-time* is used to define systems where the time taken for the execution of a task is temporally deterministic (predictable). This yields, at the task level, the notion of *hard* deadlines: a task must complete within the specified time. Thus a real-time system executes tasks in a predictable manner with respect to time.

The degree of predictability is the basis for the terminology used to describe real-time systems. Widely used categories are *hard* real time and *soft* real time. This degree-of-predictability classification conveys relative descriptive utility, but more precise definitions are implied for a given application.

In hard real-time systems, task execution that completes at an incorrect time means system failure. A missed deadline is the same as a wrong answer.

In soft real-time systems, task execution that completes at an incorrect time means reduced system performance. A missed deadline is not catastrophic, but rather degrades system performance.

Examples of hard real-time activities are:

- flight control (inertial guidance and navigation)
- nuclear power plant control
- pacemakers (human heart)
- vehicle anti-lock braking
- air-bag deployment systems

Examples of soft real-time activities are:

- command interpretation of inputs from a user interface
- saving or displaying management data
- ship navigation
- certain types of telecommunications traffic shaping functions

In general, real-time applications consist of soft and hard deadlines. Operating systems try to guarantee the individual timing constraints of the hard deadline tasks while attempting to minimize the average response times of the soft ones. Real-time operating system (RTOS) kernels achieve this through the use of appropriate features:

- near constant time system calls
- the ability to associate priority not only with the threads (or tasks) executing, but also the synchronization constructs such as mutexes
- pre-emption to achieve greater determinism
- appropriate scheduling strategies

4.1.1 Time- and Event-Triggered Systems

Another way to classify real-time systems is based on whether they are time-triggered or event-triggered. A trigger is an event that causes the start of some action, for example, the execution of a task or the transmission of a message.

There are two distinctly different approaches to the design of real-time computer applications: the event-triggered (ET) approach, and the time-triggered (TT) approach. A triggering mechanism is used to start communication and processing activities in each node of a computer system (network).

In the ET approach, all communication and processing activities are initiated upon occurrence of a significant change of state. The regular event of a clock tick is not such an event. In the TT approach, all communication and processing activities are initiated at predetermined times. While ET systems are flexible, TT systems are temporally predictable. In this guide, the systems discussed are event-triggered.

4.1.2 Developing Real-time Systems with RTOS

Real-time Operating System (RTOS) kernels are built to support real-time tasking through a number of important features that real-time systems use:

- priority based scheduling to perform real time inter-kernel process management
- priority aware synchronization constructs (semaphores for instance)
- concurrency constructs such as multi-tasking or multi-threading
- real-time clock for a time reference for internal kernel task management and housekeeping tasks
- mechanisms for inter-process and intra-process communication with associated synchronization primitives
- bounded, constant-time fast context switch, and often an associated minimal base kernel size (typically 16-32kb)

- internal kernel architecture geared to respond to external interrupts in a fast manner, and so separate their execution from intra-kernel tasks

Pre-emption and priority-based scheduling are the most important characteristics of real-time kernels. Together they give rise to the notion of priority, the central mechanism used to achieve predictable, deterministic behaviour. These characteristics are sufficient for soft real-time systems. Behavioural characteristics include quick response and small execution times for higher priority tasks - while yielding small average response times for other tasks. For hard real-time applications however, a centrally important theme is missing in such kernels. It is the notion of some form of guarantee, which is necessary for time-critical, hard real-time behaviour.

To achieve hard real-time, distributed applications, the most important properties of a distributed, mission-critical system RTOS and ORB tuple are:

- *predictability* - The RTOS must be able to predict in an *a priori* fashion the consequences of scheduling any and all tasks under its control. If it is not possible to guarantee an upper bound for the execution time of any task, the RTOS must be able to take an alternative course of action to cope with such events. Predictability is by far the single most important requirement on an RTOS, especially for hard real-time application hosting.
- *timeliness* - The RTOS must comprise internal clocks for effective handling of tasks with differing time constraints, and degree of importance or criticality.
- *fault-tolerance* - The RTOS should be immune (to some degree) to certain classes of hardware and software failures. Mission critical components in such high availability RTOS models should have fault-tolerance features inherent in their design.
- *design for peak load* - The RTOS should provide some continued minimal level of performance when subjected to unusually high peak loads. RTOS failure and crash under such circumstances is an unacceptable scenario for hard real-time applications. Therefore, they must be designed to cope with anticipated scenarios of high sporadic load.
- *maintainability* - The RTOS kernel and ORB need to be designed in a modular, pluggable fashion to ensure a minimal, optimised use of RTOS resources under any load. In addition, the ability to make modification/customisations to the kernel - as the ORB based application might require - should be minimally cross-coupled so as to be able to make the changes easily.

4.1.3 Predictability in Distributed Applications

Predictability of a complex, distributed, real-time application is achieved through the careful combination of RTOS features, networking transport, IPC mechanism implementations, and constant-time ORB internals design. A sum of these, yields a degree of predictability that enables some level of Quality of Service (QoS) to be furnished to the application built on the RTOS-ORB combination.

As far as the RTOS is concerned, it should be able to plot the evolution of tasks and events ahead of time in a given situation such that it can guarantee in advance that all critical timing constraints are met by suitable scheduling of its internals. Components that contribute to the possibility of predictably scheduling deadline-restricted tasks are:

- the features and numbers of CPUs and the scheduling policies they support
- internal CPU features such as pre-fetch, pipe lining, cache memory, and direct memory access, which can contribute to non-determinism
- types of scheduling algorithms employed in the kernel
- synchronization mechanisms
- types of priority-aware semaphore
- memory management policies, especially heap management
- communication mechanism, e.g., whether the kernel is based on messages or signals
- interrupt handling mechanisms

4.1.4 Features and Non-Determinism

It is important for the distributed real-time application designer to understand the features that will most contribute to non-determinism. These are discussed briefly in the context of an RTOS and ORB.

Probably the single greatest contributor, at the ORB level, of non-determinism is a transport that is not QoS aware or priority respecting. In essence, the management of ORB, application, and RTOS internal tasks needs to be efficiently managed by the RTOS.

Perhaps the single greatest enemy of an effective hard real-time system design is the phenomenon referred to as priority inversion.

Priority inversion occurs when a high priority task (that is, of possibly greater importance and criticality) is blocked by a less critical, lower priority thread for an unbounded period of time. This type of situation is often seen when the high priority thread is trying to get access to a shared (with the low priority task) resource, which

the low priority task has locked for its own use. There is much detailed real-time literature on this subject, and designs for its avoidance. For further reading, see *Bibliography* on page 123, particularly Rajkumar and Buttazo.

The integration of ORB and application tasks is under the control of the application designer, but the tasking and priority level control of the transport threads is not, and can give rise to priority inversions.

Other major contributors to non-determinism include:

DMA - Certain methods of direct memory - such as cycle-stealing access, used to transfer data between devices and main memory - give rise to unbounded delays. However, this can be overcome by using other techniques, such as time-slice methods.

cache - This procedure buffers CPU-RAM exchanges in an attempt to reduce task execution times. Under certain circumstances, this can contribute to non-determinism.

interrupts - These events can be sporadically triggered due to I/O devices and can impair predictability of a real time system due to the fact that they introduce unbounded delays into the execution times of other processes.

system calls - The calls for hard real-time kernel primitives need to be pre-emptible and implemented to have bounded execution times. These are then used by the scheduling subsystem of the kernel to produce the necessary guaranteed, temporally-correct behaviours internally in the kernel.

semaphores - These should be modified to be priority aware and thus avoid the priority inversion phenomenon. RTOS' normally furnish priority protocols when implementing this modification. Examples include basic priority inheritance, priority ceiling, and stack resource policy. These protocols temporarily modify task priorities to avoid deadlock and anomalous priority assignments, which cause non-determinism.

memory management - This must not produce unbounded delays in the course of execution of real-time tasks. A common practice is to use fixed, constant time type schemes to allocate, and address memory partitions to achieve predictable memory access. It is usual to see a greater degree of static allocation, which reduces flexibility for dynamic environments. The designer of real-time systems must make trade off decisions when implementing on an RTOS using languages that permit dynamic heap memory allocations, such as C++.

5 Introduction to Real-time CORBA

This chapter introduces the essential aspects of the Real-time CORBA ORB.

Please note that real-time CORBA examples are provided in the OpenFusion RTOrb Java(tm) Edition distribution's html pages.

5.1 Real-time Specification

The *Real-time CORBA Specification* defines a set of real-time extensions to standard CORBA specification.

Figure 6 shows the key Real-time CORBA entities. The features that these relate to are described below.

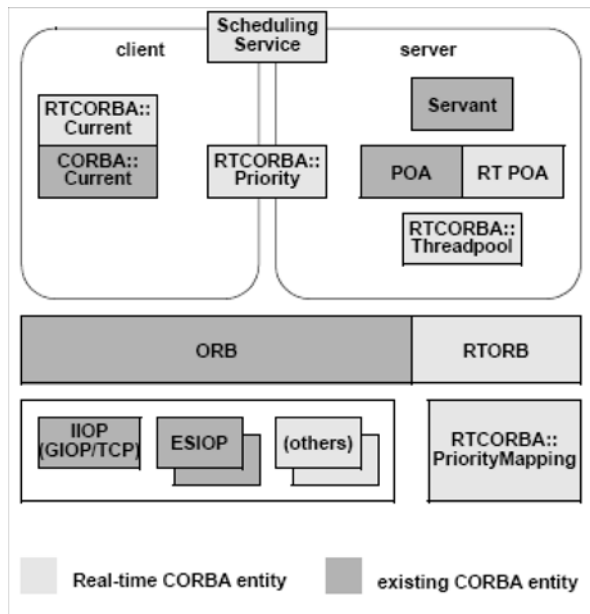


Figure 6 Real-time CORBA Extensions

5.1.1 Real-time CORBA Modules

All CORBA IDL specified by Real-time CORBA is contained in new modules `RTCORBA` and `RTPortableServer` (with the exception of new service contexts, which are additions to the IOP module.)

5.1.2 Real-time ORB

Real-time CORBA defines an extension of the ORB interface, `RTCORBA::RTORB`, which handles operations concerned with the configuration of the real-time ORB and manages the creation and destruction of instances of other Real-time CORBA IDL interfaces.

5.1.3 Thread Scheduling

Real-time CORBA uses threads as a schedulable entity. Generally, a thread represents a sequence of control flow within a single node. Threads form part of an activity. Activities are scheduled by coordination of the scheduling of their constituent threads. Real-time CORBA specifies interfaces through which the characteristics of a thread that are of interest can be manipulated. These interfaces are Threadpool creation and the Real-time CORBA Current interface. The Real-time CORBA view of a thread is compatible with the POSIX definition of a thread.

5.1.4 Real-time CORBA Priority

Real-time CORBA defines a universal, platform independent priority scheme called Real-time CORBA Priority. It is introduced to overcome the heterogeneity of different Operating System provided priority schemes, and allows Real-time CORBA applications to make prioritised CORBA invocations in a consistent fashion between nodes with different priority schemes.

For consistency, Real-time CORBA applications always should use CORBA Priority to express the priorities in the system, even if all nodes in a system use the same native thread priority scheme, or when using the server declared priority model.

5.1.5 Native Priority and Priority Mappings

Real-time CORBA defines a `NativePriority` type to represent the priority scheme that is 'native' to a particular Operating System.

Priority values specified in terms of the Real-time CORBA Priority scheme must be mapped into the native priority scheme of a given scheduler before they can be applied to the underlying schedulable entities. On occasion, it is necessary for the reverse mapping to be performed in order to obtain a Real-time CORBA Priority to represent the present native priority of a thread. The latter can occur, for example, when priority inheritance is in use or when wishing to introduce an already running thread into a Real-time CORBA system at its present (native) priority.

Real-time CORBA defines a `PriorityMapping` interface in order to allow the Real-time ORB and applications to do both of these things.

5.1.5.1 User-defined PriorityMappings

PrismTech provides a priority mapping implementation, *com.primstech.ofj.rtorb.PriorityMapping*, which is automatically instantiated when the ORB is initialised. This default implementation uses a linear algorithm that maps a range of CORBA priority values to the range `MIN_NATIVE_PRIORITY` to `MAX_NATIVE_PRIORITY`. This mapping algorithm is identical to the algorithm used by TAO.

The priority mapping can be retrieved using the following static accessor call on `com.primstech.ofj.rtorb.RTORB`:

```
public static org.omg.RTCORBA.PriorityMapping
    com.primstech.ofj.rtorb.RTORB._get_priority_mapping()
```

A developer can provide their own priority mapping implementation. User-defined priority mapping implementations can be used by calling the following static method on `RTORB`:

```
public static void
    com.primstech.ofj.rtorb.RTORB._set_priority_mapping(
        org.omg.RTCORBA.PriorityMapping mapping)
```

This method overrides the default implementation with the supplied version.

5.1.6 Real-time CORBA Current

Real-time CORBA defines a Real-time CORBA Current interface to provide access to the CORBA priority of a thread.

5.1.7 Priority Models

One goal of Real-time CORBA is to bound and to minimize priority inversion in CORBA invocations. One mechanism that is employed to achieve this is propagation of the activity priority from the client to the server, with the requirement that the server side ORB make the up-call at this priority (subject to any priority inheritance protocols that are in use).

However, in some scenarios, it is sufficient to design the application system by setting the priority of servers, and having them handle all invocations at that priority. Hence, Real-time CORBA supports two models for the priority at which a server handles requests from clients:

- **Client Propagated Priority Model:** in which the server honours the priority of the invocation, set by the client. The invocation's Real-time CORBA Priority is propagated to the server ORB and the server-side ORB maps this Real-time CORBA Priority into its own native priority scheme using its `PriorityMapping`.

Requests from non-Real-time CORBA ORBs; that is, ORBs that do not propagate a Real-time CORBA Priority with the invocation are handled at a priority specified by the server.

- **Server Declared Priority Model:** in which the server handles requests at a Real-time CORBA Priority assigned on the server side. This model is useful for setting a boundary where new activities are begun with a CORBA invocation.

5.1.8 Real-time CORBA Mutexes and Priority Inheritance

The Mutex interface provides the mechanism for coordinating contention for system resources. Real-time CORBA specifies an `RTCORBA::Mutex` locality constrained interface, so that applications can use the same mutex implementation as the ORB.

A conforming Real-time CORBA implementation must provide an implementation of `Mutex` that implements some form of priority inheritance protocol. This may include, but is not limited to, simple priority inheritance or a form of priority ceiling protocol. The mutexes that Real-time CORBA makes available to the application must have the same priority inheritance properties as those used by the ORB to protect resources. This allows a consistent priority inheritance scheme to be delivered across the whole system.

5.1.9 Threadpools

Real-time CORBA uses the Threadpool abstraction to manage threads of execution on the server-side of the ORB. Threadpool characteristics can only be set when the threadpool is created. Threadpools offer the following features:

- *preallocation of threads* - This helps reduce priority inversion, by allowing the application programmer to ensure that there are enough thread resources to satisfy a certain number of concurrent invocations, and helps reduce latency and increase predictability, by avoiding the destruction and recreation of threads between invocations.
- *partitioning of threads* - Having multiple thread pools associated with different POAs allows one part of the system to be isolated from the thread usage of another, possibly lower priority, part of the application system. This can again be used to reduce priority inversion.
- *bounding of thread usage* - A threadpool can be used to set a maximum limit on the number of threads that a POA or set of POAs may use. In systems where the total number of threads that may be used is constrained, this can be used in conjunction with threadpool partitioning to avoid priority inversion by thread starvation.
- buffering of additional requests beyond the number that can be dispatched concurrently by the assigned number of threads.

5.1.10 Priority Banded Connections

In order to reduce priority inversion due to use of a non-priority respecting transport protocol, RT CORBA provides the facility for a client to communicate with a server via multiple connections, with each connection handling invocations that are made at a different CORBA priority or range of CORBA priorities. The selection of the appropriate connection is transparent to the application, which uses a single object reference as normal.

5.1.11 Non-Multiplexed Connections

Real-time CORBA allows a client to obtain a private transport connection to a server, which will not be multiplexed (shared) with other client-server object connections.

5.1.12 Invocation Timeouts

Real-time CORBA applications may set a timeout on an invocation in order to bound the time that the client application is blocked waiting for a reply. This can be used to improve the predictability of the system.

5.1.13 Client and Server Protocol Configuration

Real-time CORBA provides interfaces that enable the selection and configuration of protocols on the server and client side of the ORB.

5.1.14 Real-time CORBA Configuration

New policy types are defined to configure the following server-side RT CORBA features:

- server-side thread configuration (through Threadpools)
- priority model (propagated by client versus declared by server)
- protocol selection
- protocol configuration

Which CORBA policy application points (ORB, POA, Current) that a given policy may be applied at is given along with the description of each policy. Real-time CORBA defines a number of policies that may be applied on the client-side of CORBA applications. These policies allow:

- the creation of priority-banded sets of connections between clients and servers;
- the creation of a non-multiplexed connection to a server;
- client-side protocol selection and configuration.

In addition, Real-time CORBA uses an existing CORBA policy to provide invocation timeouts.

5.2 Real-time Portable Object Adapters

Real-time Portable Object Adapters (RTPOA) configuration is one of the most important features in real-time CORBA. Application developers can configure and control hardware resources using real-time policies associated with real-time POAs.

This section describes priority models, the pluggable RTPOA, threads and threadpools, and priority banded connections.

5.2.1 Priority Model

RTOrb only supports both the `RTCORBA::SERVER_DECLARED` and `RTCORBA::CLIENT_PROPAGATED` priority models. Refer to the CORBA Priority Model example included in the RTOrb examples to see how to set the `RTCORBA::SERVER_DECLARED` priority model policy for an RTPOA.

5.2.2 RTPOA

The RTPOA module which extends the standard POA interface with respect to priority and resource configuration.

5.2.2.1 POA Activation Methods with Priority

```
create_reference_with_priority()
create_reference_with_id_and_priority()
activate_object_with_priority()
activate_object_with_id_and_priority()
```

5.2.3 Threads and Threadpools

There are two basic ways of manipulating threads in RT CORBA, `RTCORBA::Current` and Threadpools (via policies at POA creation time).

5.2.3.1 Current

RT CORBA defines a `RTCORBA::Current` interface to provide access to the CORBA priority of a thread. Please refer to the CORBA Priority example included with this product on how to access the priority of a thread.

5.2.3.2 Threadpools

Thread pools are one of the most important features in Real-time CORBA. Threads in pools can be pre-allocated and partitioned amongst active Real-time POA's. Application developers and end-users configure and control processor resources using thread pools. The possibility of experiencing priority inversion can be bounded and reduced by configuring real-time POA's with threadpools where each POA associates with one or more thread pools (see *Figure 7*). Note that threadpools are independent of the POA lifecycle.

5.2.3.3 Thread Pool Operation Basic Mode

Application developers and end-users configure and control processor resources using thread pools (see *Figure 7*). Threads in the threadpool execute requests at the object priority for which each request is targeted. Each POA associates with one or more thread pools. However, you are reminded that thread pools are independent of the POA lifecycle.

To dispatch requests to the correct queue and to the right servant on the server side, each request needs to be handled by the right priority thread. To achieve this, requests are pushed onto the queue of appropriate priority and are processed synchronously by the waiting threads within a lane. There is a queue assigned to each thread pool.

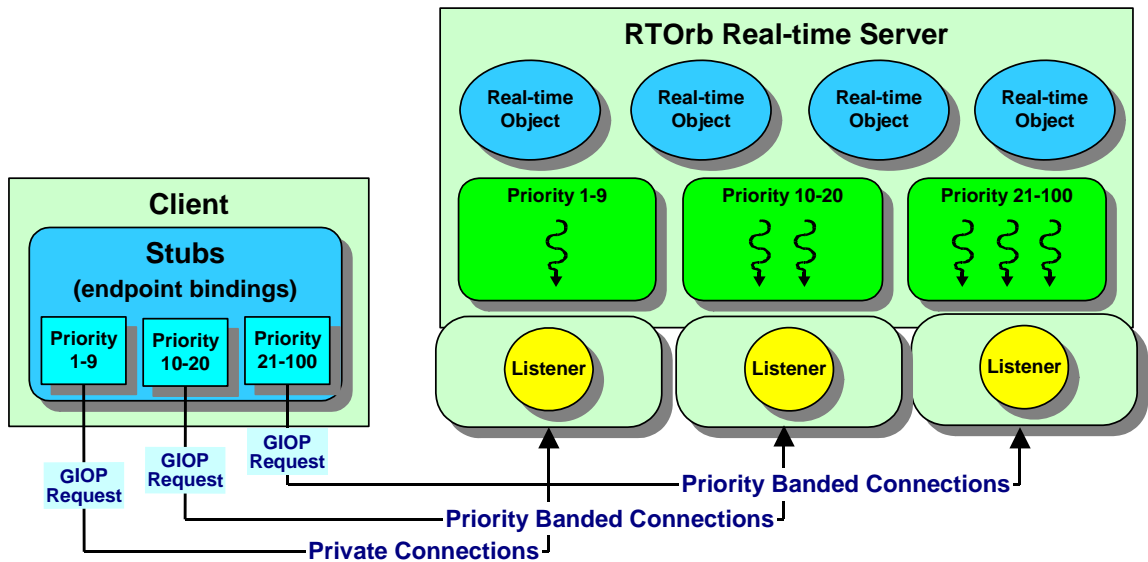


Figure 7 Controlling Network Resources

The client side may hold multiple connections open through the use of individual object references to end points in the server, based on priority band.

Threadpools can be configured for use with RTPOA's in one of two forms:

1. Non-laned Threadpool
2. Laned Threadpool

5.2.3.4 Laned Threadpool

A threadpool can be created that has n partitions (lanes) each created to serve requests at a specific priority. Each lane has m static threads running at the priority defined for the lane. Whenever a request arrives, a lane is chosen based on the priority associated with the activated object. Please refer to the Threadpools example included with this product on how to create a Threadpool with lanes.

As seen above, the half-sync layer consists of a thread pool associated with a POA. A thread pool can be shared among POAs.

5.2.3.5 Priority Banded Connections

RT CORBA introduces the concept of priority banded connections. A real time POA (RTPOA) supporting priority banded connections is capable of accepting requests across transport with some concept or awareness of the requestors priority at which the server should execute. Each client can open a number of connections with a server, each connection handling a range of priorities defined in the priority banded connection policy.

Priority banded connections are useful when used in conjunction with a transport protocol that does not respect priorities. Transports like TCP that are not easily pre-emptable and do not respect priorities can incur head of line blocking where requests of higher priority are blocked and unable to pre-empt requests at lower priority. This leads to unbounded delays and the potential of priority inversion. Priority bands allow multiple connections to be utilized to minimize the head of line blocking that can occur where one connection is used for multiple priority requests.

An RTPOA that is configured with laned threadpools and priority banded connections can provide more predictability. Please refer to the Connections example included with this product on how to create priority banded connections.

5.2.4 RTPOA Current

This interface is available to perform operations to access the identity of the object on which a call was invoked. This is supplied for supporting servants that may implement multiple objects.

5.2.5 Associations Between Pools and RTPOA

Each POA must have at least one thread pool attached to it. This is done by passing a thread pool policy to the POA. In the case where no policy is specified or an invalid threadpool identifier is used, the ORB will use the default non-real time threading approach, which consists of unlimited dynamic thread allocation. One thread pool can be shared among multiple POAs.

5.3 Priority Machinery

Priority is the medium used to achieve QoS in real-time CORBA, hence the focus of RTOrb application design. With the RTOrb priority scheduling is achieved via the RTOS scheduler. Tasks or threads that comprise the application execute in a stable, predictable manner as a result of this priority scheduling. In addition if using only the RTOS for scheduling purposes, it must provide proper mutexes and semaphores to resolve resource contention, such as priority-aware application objects and/or code segments.

The central theme in real-time CORBA programming is the notion of prioritised scheduling of activities, tasks, or threads.

This section provides:

- background information on the phenomenon of priority inversion
- discussion of protocols used to overcome priority inversion
- discussion of priority mapping and CORBA priority scheme

5.3.1 Priority Phenomena and Protocols

Priority inversion is a commonly known phenomenon in real-time systems. It usually manifests in the form of unbounded delays of high priority tasks. Normally, when priority inversion occurs, high priority tasks are forced to wait on low priority tasks. This occurs when the high priority tasks are sharing common resources with low priority tasks. If a low priority task locks the resource for its own use but is pre-empted by a higher priority task, which also needs access to the common resource, the high priority task will have to wait on the lower priority task.

To illustrate the concept of priority inversion more clearly, consider *Figure 8*. Here, 3 tasks or threads are executing, T_1 , T_2 , T_3 . The tasks are illustrated in order of decreasing priority such that the priority of T_1 is the greatest and that of T_3 the least of the 3. In addition, we assume that T_1 and T_3 share a common resource, such as a critical section, to which only one can have access at any point in time. The following is a typical scenario illustrating priority inversion.

At time t_0 task T_3 starts to run. At time t_1 task T_3 locks and enters a critical section, continuing to execute until time t_2 . The portion of time for which task T_3 is in a critical section is shown as shaded. At time t_2 , task T_1 pre-empts task T_3 because T_1 has a higher priority. Task T_1 now executes from time t_2 until time t_3 , at which point it attempts to gain access to the critical section, which has previously been locked by lower priority task T_3 . Task T_1 is therefore forced to wait or block until such time as T_3 releases the lock on the critical section shared between T_1 and T_3 . Task T_3 is allowed to run next. So at time t_3 , task T_3 resumes execution and continues to work its way through the critical section.

Now at time t_4 , task T_2 pre-empts task T_3 and starts to run because task T_2 has a higher priority than that of task T_3 .

Task T_1 is now blocked by task T_3 because of the shared resource, and task T_3 is blocked by task T_2 . Therefore T_2 is now indirectly blocking task T_1 as well. Task T_2 blocks task T_3 until T_2 completes at time t_5 . As a consequence T_3 is forced to block for a significant amount of time (the length of the shared critical section plus the execution time of task T_2).

For an actual system, when several medium priority tasks exist with priorities greater than that of task T_3 but less than that of task T_1 , it can lead to unbounded delay or blocking.

This effect is known as priority inversion and occurs in the time interval t_3 to t_6 .

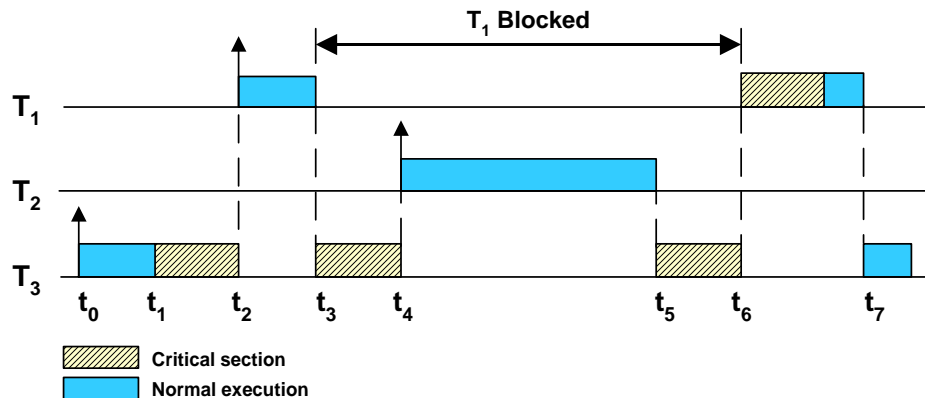


Figure 8 Priority Inversion

The priority inversion phenomenon in real-time systems is one that can manifest any time several tasks want to execute in the presence of services that are shared among them.

Several approaches have been proposed to alleviate the priority inversion phenomenon in real-time systems and much literature is available. A complete description and analysis is beyond the scope of this document. The reader is directed to further reading under *Bibliography* on page 123, particularly Buttazo.

The Real-time Extension aids the RT CORBA developer by providing priority inheritance protocols in the ORB. Specifically, RTOrb's RT CORBA mutex supplies a default implementation that uses the simple priority inheritance protocol as an example. Other protocols are also possible, but this is used to illustrate the concept and its applicability.

The priority inheritance protocol bounds any priority inversion that could possibly occur. Although the ORB's initial design is such that it tries to eliminate the possibility, it can still occur as a result of unusual transports, or hardware specifics that are used in a particular setup.

Figure 9 and the following text explain how priority inheritance protocols bound any possible priority inversion. The same three tasks are illustrated as in *Figure 8*. Additionally, the relative priorities of the three tasks are depicted at the bottom of the figure as P_1 , P_2 , and P_3 .

Up to time t_3 , the behaviour of tasks T_1 and T_3 are the same as in *Figure 8*. At time t_3 , T_1 is forced to block on T_3 due to T_3 holding a lock on a critical section to which T_1 needs access. At this point the mechanism of priority inheritance is employed. This mechanism causes T_3 to inherit the priority P_1 of task T_1 , which forces task T_3 to execute immediately and run through the remaining part of its critical section. This forces T_3 to execute from t_3 to t_5 at the T_1 priority P_1 , which is the highest priority in this illustration. Note that task T_2 cannot pre-empt task T_3 as task T_2 has lower priority than the temporarily assigned priority (P_1 of task T_3 , through priority inheritance).

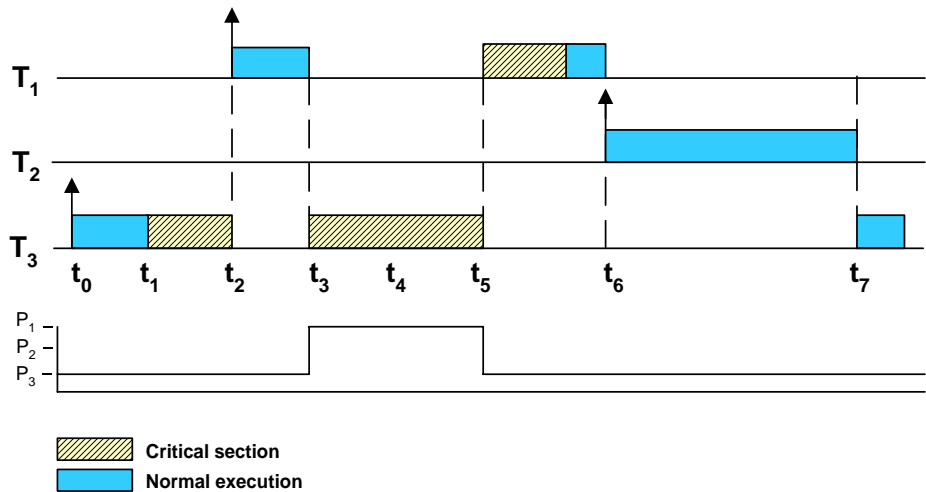


Figure 9 Priority Inheritance Protocol to Bound Priority Inversion

As task T_3 exits the critical section, its priority is returned to its original value P_3 as shown in *Figure 9*. At time t_5 , task T_1 can run because priority P_1 is greater than priority P_2 of task T_2 . Thus it no longer needs to block on task T_3 , which was holding a lock on the critical section. Task T_1 now runs through the critical section and to completion at t_6 . At time t_6 , task T_2 has the highest priority and executes as shown in *Figure 9*.

5.3.1.1 CORBA Priority

CORBA uses a standard (canonical) form of priority that can be mapped to any RTOS priority scheme. In effect, CORBA subsumes the heterogeneity in RTOS-specific priority schemes and thus achieves uniformity. This allows CORBA invocations to be made across multiple, different RTOS platforms - which may have different native priority schemes - in a consistent manner. Therefore, CORBA priority is a wrapper for native priority schemes.

5.3.1.1.1 CORBA Priority Mapping

Priorities may be mapped from the CORBA priority scheme to the RTOS native priority scheme. This is accomplished with an interface defined in IDL, and allows you to forward and reverse map CORBA and native priorities as shown in *Figure 10*.

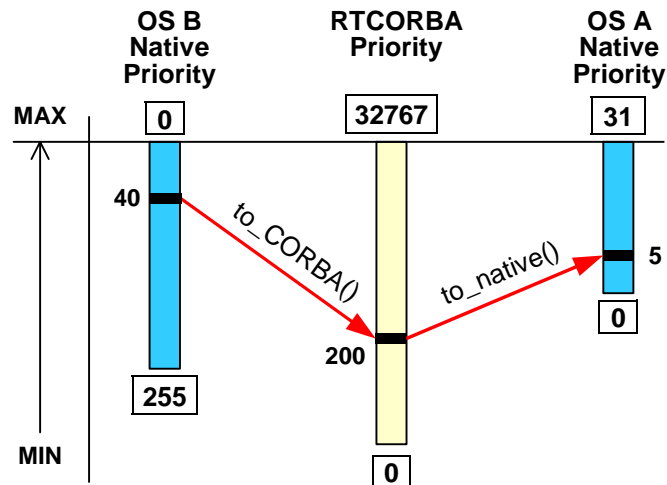


Figure 10 Priority Mapping

An RTCORBA priority type id, defined in IDL to be of type CORBA short, is as follows:

```
module RTCORBA {
  typedef short Priority;
  const Priority minPriority = 0;
  const Priority maxPriority = 32767;
};
```

It spans the interval 0 to 32767. Higher values of RTCORBA priorities map to higher native RTOS priorities.

5.3.1.1.2 RTCORBA Current Interface

The `Current` interface in RTCORBA allows a developer access to the priority data of the current locus of execution or thread. The interface allows for setting and getting a thread's CORBA priority.

```
interface Current : CORBA::Current {
    attribute Priority the_priority;
};
```



A thread has native base and elevated priorities, which may be different than the observed CORBA mapped value.

This is a local interface, which also stores information about its current CORBA and native priorities in a thread-local storage structure. It is a singleton within the context of its present locus of execution. A typical application's use of the RTCORBA current interface is illustrated below: Please refer to the CORBA Priority example included with this product on how to use the `RTCCurrent` get and set methods, and use of the default priority mapping.

5.4 CORBA Mutex

5.4.1 Mutex Notifies in RT CORBA

Real-time CORBA specifies an `RTCORBA::Mutex` locality-constrained interface so that applications can use the same mutex implementation as the ORB. This mutex interface provides the mechanism popularly used to coordinate access to shared resources. In RTCORBA such a construct is required to have associated with it a priority inheritance protocol to resolve any resource access contention by threads of differing priorities.

5.4.2 Why Mutex Has a Priority Protocol

The RTCORBA specification requires a mutex implementation to have some form of priority inheritance protocol. This may include, but is not limited to, simple priority inheritance. In addition, any type of priority-aware mutex that the ORB makes available to the application must have the same priority inheritance protocols as those used by the ORB to protect its own internal resources. It is imperative to eliminate, if not bound, the priority inversion phenomenon, thereby allowing for consistency across the whole system with regard to resolving any resource access contention.

5.4.3 The Real-time CORBA Mutex Interface

The IDL for the real-time CORBA specification is defined as:

```
module RTCORBA
{
    interface Mutex
    {
        void lock ();
        void unlock ();
        boolean try_lock (in TimeBase::TimeT max_wait);
    };

    interface RTORB
    {
        ...
        Mutex create_mutex ();
        void destroy_mutex (in Mutex the_mutex);
        ...
    };
};
```

6 Introduction to Real-time Java

The Real-Time Specification for Java (RTSJ) defines an extension to the Java specification for creating real-time applications using the Java programming language. Real-Time Java (RTJ) has seven areas of extended semantics, including thread scheduling and dispatching, memory management, synchronization, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination and physical memory access.¹

This chapter provides high level descriptions of these areas and their constituent components. You are directed to the Real-Time Specification for Java for complete descriptions and the semantics that apply to particular classes, constructors, methods and fields. It is strongly recommended that you refer to the references listed under Further Reading and Examples Information on page 74.

6.1 Real-time Extension to Java

6.1.1 Thread Scheduling and Dispatching

Real-time programming must ensure that the execution of machine instruction sequences, such as a *thread* for example, are timely² or predictable. RTSJ uses the concept of *schedulable objects* and *schedulers* to schedule the execution of these instruction sequences.

Scheduling is the arranging of a set of threads to be executed in a particular order. The order of thread execution is called a *schedule* and attempts to optimise the system's ability to meet (predefined) temporal constraints. The schedule's method of optimising the thread execution is called its *metric*. For example, a typical metric in hard real-time systems is "number of missed deadlines", noting that the only acceptable value for that metric in a *hard* real-time system is zero (recalling that hard real-time system do not allow time deadlines to be overrun.).

A schedulable object is implemented in RTSJ as an instance of any class that implements the *Schedulable* interface. Three examples of schedulable objects are RTJ's *RealtimeThread*, *NoHeapRealtimeThread*, and *AsyncEventHandler*.³

-
1. The RTSJ specifies seven, however other sources name eight areas of extended semantics, *exceptions* being the eighth.
 2. The term *timely* here means that a thread's execution will always complete before a given time limit has been passed.
 3. Each of these three objects are required in any RTSJ-based application.

A scheduler is implemented as an instance of *class Scheduler*. Scheduling and dispatching of the schedulable object is managed by a *Scheduler* instance. The schedulable object has a reference to the *Scheduler* instance.

There are often situations where schedulable objects compete for the resources so they can run. All schedulable objects run in a thread. Each thread is assigned a numeric priority value. This value is referred to as the thread's *priority*. The scheduler uses a thread's priority to help determine the thread's execution eligibility: threads are selected for execution in the order of their priority, highest to lowest, with the confines of a scheduling algorithm. The term *dispatching* refers to selecting a thread with the highest execution eligibility, in other words the highest priority, from the pool of threads that are ready to run.

The programmer is responsible for assigning thread priorities.

6.1.2 Memory Management

RTJ addresses the issue of the unpredictable latencies of Java's Garbage Collector by providing memory management facilities which do not interfere with the deterministic behaviour of real-time code. RTJ uses *memory area types* to enable programmers how memory is allocated, controlled and released or garbage collected.

RTJ has four basic memory area types. These are:

1. *Scoped memory* - gives bounds to an object's lifetime of objects on the heap (syntactic scope). When a scope is entered, every use of *new* causes the memory to be allocated from the active memory scope. Scoped memory types provides flexibility by allowing the application to use a memory area that has characteristics that are appropriate to a particular syntactically defined region of the code.
2. *Physical memory* - objects are created within specific physical memory regions that have particular characteristics, such as having substantially faster access than other memory locations.
3. *Immortal memory* - objects remain in memory from the point they are allocated until the application's Java runtime environment terminates. It is a memory resource shared among all threads in an application. Objects allocated to immortal memory are never garbage collected or moved.
4. *Heap memory* - represents the heap. RTSJ does not change the determinant of lifetime of objects on the heap: lifetime is determined by visibility.

Some support is provided by RTJ for budgeting memory allocation for threads using memory areas. Maximum memory area consumption and maximum allocation rates for individual real-time threads may be specified when the thread is created.

6.1.3 Synchronization

Serialized resources in a real-time environment can be subject to priority inversion¹. Priority inversion is dealt with by RTJ through:

- algorithms that prevent priority inversion between real-time Java threads when they share a serialized resource (such as when the Java *synchronized* keyword is used)
- providing wait-free queue classes which can be used when the priority inversion-protection algorithms are insufficient

6.1.4 Asynchronous Event Handling

RTJ provides efficient mechanisms for programming disciplines that can accommodate the asynchronous behaviour which exists in the real world. RTJ generalizes the Java language's mechanism of asynchronous event handling. A notable feature of RTJ's asynchronous event handling is that the execution of an application's logic is scheduled and dispatched by an implemented scheduler.

6.1.5 Asynchronous Transfer Of Control

There are instances when the real-world environment changes both drastically and asynchronously which requires the current point of logic execution to be immediately and efficiently transferred to another location. RTSJ has a mechanism which extends Java's exception handling to allow applications to programatically change the point of control of another Java thread. Note that this asynchronous transfer is restricted to specifically written control logic which assumes that its point of control may change asynchronously.

6.1.6 Asynchronous Thread Termination

Application logic may need to arrange for a real-time Java thread to expeditiously and safely transfer its control to its outermost scope and end in a normal manner when occasional drastic and asynchronous changes in the real-world happen. RTJ's mechanism for asynchronous event handling and transfer of control is safe, unlike the (deprecated) traditional unsafe Java mechanism for stopping threads.

6.1.7 Physical Memory Access

Physical memory access is desirable for many applications. RTJ provides a class that allows programmers byte-level access to physical memory, as well as a class that allows objects to be constructed in physical memory.

1. Priority inversion occurs, for example, when a high priority task is waiting for a resource which has been locked by a lower priority task, thereby preventing the high priority task from running. This is considered to be a system failure.

6.2 Further Reading and Examples Information

Real-time Java examples are provided in the OpenFusion RTOrb Java(tm) Edition distribution's html pages.

A recommended reading and reference list is given in the *Bibliography* on page 123. For convenience, references which are particularly relevant to RT Java are listed below.

Real Time Java Platform Programming, Peter C. Dibble, Sun Microsystems Press Java Series, 2002.

Highly recommended, provides essential information.

Concurrent and Real-Time Programming in Java, by Andy Wellings, John Wiley & Sons Ltd, 2004.

This book provides an introduction to concurrent and real-time programming, describes Java's concurrency model, introduces and discusses the RTSJ, and includes examples.

Real-Time Systems and Programming Languages, Alan Burns and Andy Wellings, Addison Wesley, Third Edition, 2001.

Describes real-time programming in Ada 95, Java, occam2 and C; covers a variety of associated topics including, for example, reliability, concurrent programming, design and distributed systems.

A close-up, low-angle photograph of a computer keyboard, showing several keys in detail. The keys are white and the keyboard is dark. A white grid pattern is overlaid on the image, creating a sense of depth and perspective. The text "PROGRAMMING WITH RTO RB" is centered in the upper half of the image.

PROGRAMMING WITH RTO RB

7

Using the ORB

RTOrb can be used to create CORBA-based client-server applications, for either enterprise or real-time applications. The major steps of creating applications using RTOrb are the same whether creating enterprise or real-time applications, even though the internal details of their respective client and server classes differ.

7.1 Introduction

The steps for creating RTOrb-based applications are:

- Step 1:** Declare the application's classes and/or interfaces in an IDL specification. (See *About this release of OpenFusion RTOrb Java(tm) Edition* on page 78).
- Step 2:** Compile the application's IDL specification with the IDL compiler to create the Java source files and classes for the stubs, skeletons and/or tie classes and interfaces.
- Step 3:** Write an implementation for the interface generated from the IDL specification.
- Step 4:** Write a server (if not being implemented by third parties) as a *main* class. The main class instantiates the server as well as starts running the ORB.
- Step 5:** Write a client (if not being implemented by third parties) as a *main* class. The client makes (remote) requests to the server instance.
- Step 6:** Compile the developer-written source code and IDL-generated source code with a Java compiler for the required platform.
- Step 7:** Deploy the application's server and client components on the required platforms.
- Step 8:** Start the server and run the client.

This section describes the procedures, requirements and practical details needed to create CORBA-based applications with RTOrb using the above steps. (Please note that this section is **not** intended as a tutorial of how to write CORBA-based applications. Basic information for writing CORBA-based applications is provided in Section 8, *Creating Applications*, on page 89.)

The topics covered here include:

- a general description of how to use the ORB's IDL to Java compiler
- the requirements, procedures and settings for creating, compiling, deploying and running RTOrb-based applications
- brief information about running OpenFusion CORBA Services



About this release of OpenFusion RTOrb Java(tm) Edition

- RTOrb must be initialised in immortal memory when running the ORB in hard real-time mode. A *Simple Soft Real-time Application* on page 94 shows how to initialise the ORB in immortal memory.
 - RTOrb must be initialised in heap memory when running the ORB in soft real-time mode.
 - The ability to initialise RTOrb in scoped memory is not supported in the release.
 - There is a slight memory leak when multiple calls are made to `ORB.init()`.
-

7.1.1 Advice Notes

1. The CORBA `release()` method has now been implemented on `Object`. It is recommended that developers use this method for freeing objects that are no longer needed. Refer to the *CORBA Specification*, release 04-03-12, *Section 4.3.2.2*.

2. Garbage collection can be minimised by caching structs and unions in the skeleton classes. Using the object cache is a two stage process:

- Run the idl compiler with the `-cacheplugin` argument to generate object caching code, for example:

```
idl -d generated -cacheplugin com.primstech.ofj.util.objectcache.  
    ObjectCachePluginImplementation  
myIdl.idl
```

This generates additional code in the structs' and unions' helper classes and in the skeleton classes of interfaces.

- Configure RTOrb to use object caches for specific classes by setting the `cachedObjectClasses` properties in the `ofj.properties` file. See the *Section 2.1.2, Configuration Properties*, on page 20, for information about the `objectcache` properties and their values.

The RTOrb object cache implementation enables the default implementations of `ObjectCache` and `ObjectCachePlugin` to be easily replaced with a custom object cache implementation.

3. Appendix A, *API Enhancements*, describes RTOrb enhancements to the OMG's standard CORBA API.



4. **WARNING:** There is a potential problem with Realtime Threads and static initialisers when using the *Sun Real Time Java System*. If an application is started using deeply nested Realtime Threads and `Runnable`s, and the first reference to OpenFusion Java ORB is within those Threads, then it is possible

that class static initialisers will not be created correctly. PrismTech advises that customers call `com.prismtech.ofj.util.ClassInitializer.init()` at the start of their application in order to work around this problem.

7.1.2 Conventions

The following convention is used in this section:

`<OFJ_DIR>` - the directory where RTOrb is installed

`<JAVA_HOME>` - root directory of the Java Virtual Machine (JVM) installation

7.2 Using the IDL Compiler

Basic instructions for using RTOrb's IDL to Java compiler are given here. This release of RTOrb uses the *JacORB* idl to Java compiler.

Detailed information, including descriptions of all of the compiler's command line options are provided in the *ORBUtilities.pdf* document.



You should read the instructions provided below and in the *ORBUtilities.pdf* document before attempting to use the IDL compiler. Also, ensure the `<OFJ_DIR>/bin` directory is in the system's PATH.

The compiler is run using the *idl* script located in the `<OFJ_DIR>/bin` directory.

The *idl* compiler script is used from the command line as:

```
% idl [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 *idl* with the *-h* or *-help* option displays usage information. The complete list of command-line parameters is described in *ORBUtilities.pdf*: refer to the instructions in that guide before using the *idl* compiler.

The IDL compiler creates the files listed below. The number, type and names of the generated output files can be changed using the command-line options, such as disabling the creation of POA skeletons or client stubs.

The standard types of generated source Java files are:

- *Operations* - contains the generated Java interface which is mapped from the developer-written IDL interface specification. The IDL specification includes the interface's IDL type, operations and exception definitions.

- *Interface* - contains an interface which extends the interface in the *Operations* file as well as extending `org.omg.CORBA.Object` and `org.omg.CORBA.portable.IDLEntity`. Clients are able to obtain references to objects that implement this interface.
- *Helper* - provides a helper class which is needed to manipulate the mapped IDL interface. The class contains several essential static methods, including `narrow()`.
- *Holder* - contains a class which enables *inout* and *out* parameters to be passed in Java (Java can only pass arguments by value).
- *Stub* - is responsible for delegating shared functionality, such as `isa()`, to the vendor specific implementation. This file contain a class which extends `org.omg.CORBA.portable.ObjectImpl`.
- *POA* - contains a servant base class that implements the Java *Operations* interface and provides skeleton code for the developer written object implementation. The class in the POA file extends `org.omg.PortableServer.Servant`.
- *POATie* - this file contains a class which extends the *POA* class. This class enables a developer-written implementation class to inherit from more than one servant base class, if needed, by delegating to the implementation class. This class overcomes the single inheritance restriction of the Java language.

The generated filenames for the above file types, unless customised by using the IDL compiler's command line options, are constructed as follows:

- With the exception of the *Interface* type, the interface name contained in the IDL specification file is prepended to the file's type and given a `.java` extension. For example, if using an IDL specification file called `MyApp.idl` containing an interface called `MyAppsInterface`, the generated *Operations* file would be called `MyAppsInterfaceOperations.java`.
- Files generated for the *Interface* type have the same name as the interface name defined in the IDL filename, but with a `.java` extension. For example, if a file called `MyApp.idl` contains an interface `MyAppsInterface`, then `MyAppsInterface.java` will be generated.
- The generated *Stub* files have an added underscore (`_`) prepended to the filename. For example, the *Stub* file for `MyApps.idl` containing the `MyAppsInterface` interface would be called `_MyAppsInterfaceStub.java`.

Example

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

```
% idl MyApp.idl
```

The generated files are:

```
MyAppsInterfaceOperations.java
MyAppsInterface.java
MyAppsInterfaceHelper.java
MyAppsInterfaceHolder.java
_MyAppsInterfaceStub.java
MyAppsInterfacePOA.java
MyAppsInterfacePOATie.java
```

An *ofjdefs.idl* file is located in the `<OFJ_DIR>/idl/omg` directory. This file is included in all entry point IDL files, such as, *PortableServer.idl*, to ensure that developers use the same IDL compiler options as those which were used to generate the core stubs. To ensure that the same IDL compiler options are used, pass `-DOFJ` as an argument to the IDL command line command.

7.3 Compiling Applications

The common requirements, procedures and settings needed for creating and compiling RTOrb-based applications are described below.

7.3.1 System and Environment Settings

The system's environment variables should be set as described under Section 1.1.2, *System Variables*, on page 10, before running RTOrb or RTOrb-based applications.

i RTOrb classes and all of the precompiled examples are held in *ofj.jar*. If a developer uses the supplied scripts to run applications and follows the instructions in Section 7.3.2, *Java Compiler* to compile, then it is *not* necessary to add *ofj.jar* to the classpath.

7.3.2 Java Compiler

7.3.2.1 Common Requirements

RTOrb-based applications must be compiled with a supported Java compiler. See the RTOrb *Release Notes* for a list of supported Java compilers.

i RTOrb uses *endorsed directories*. Detailed information about these is available on Sun's web site at <http://java.sun.com/j2se/1.5.0/docs/guide/standards/index.html>.

7.3.2.2 Sun Java Real-Time System Requirements

The only requirement for the Sun Java Real-Time System is that the *PATH* must include the directory where it is installed.

Example Environment Variable Setting using the Java Real-Time System with RTOrb

Where:

`/opt/myOFJ` is the directory where RTOrb is or will be installed and

`/usr/local/j9rt` is the directory where the Java Real-Time System is installed.

UNIX

```
% PATH=/opt/myOFJ/bin:/usr/local/j9rt:$PATH
% export PATH
% CLASSPATH=.
% export CLASSPATH
```

Example Compiling with the Java Real-Time System

Compile a file called `myFile.java` using the Java Real-Time System.

```
% javac -endorseddirs <install-dir>/lib/endorsed <*.java>
```

7.3.2.3 IBM Websphere Real Time JVM

The requirements which are specific to the IBM Websphere Real Time JVM are:

- Use the `javac -Xrealttime` option to enable the RTSJ classes provided by the IBM WebSphere Real Time JVM to be used.
- Compile a file called `myFile.java` with the IBM Websphere Real Time JVM:

```
% javac -Xrealttime -endorseddirs <install-dir>/lib/endorsed
<*.java>
```

7.4 Deploying and Running Applications

Information that is common to all RTOrb supported platforms about deploying and running applications is provided below.



Refer to Chapter 2, *Configuration* for information on configuration and property settings which may be needed to deploy and run RTOrb-based applications.

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.

7.4.1 RTOrb Run Scripts

OpenFusion RTOrb Java(tm) Edition provides two convenience scripts which can be used to run RTOrb-based non real-time and real-time applications, respectively `run` and `runrt` located in the `<OFJ_DIR>/bin` directory.

- The *run* script starts *non real-time* applications along with OpenFusion CORBA services (such as the OpenFusion Naming Service).
- The *runrt* script starts *soft* and *hard real-time* applications.

The *runrt* script is executed from the command line using:

```
% runrt [<defs>] <class> [<args>]
```

where

<defs> are property definitions of the form *-Dname=value*

<class> is the name of the class to run

<args> are arguments required by the class (optional)

Table 9 describes the *run* and *runrt* scripts command line options.

Table 9 *run* Script Command Line Options

Option	Description
-d	Enable debugging of OpenFusion services.
-s	Enable security controls.
-x	Use the <i>bootclasspath</i> . The <i>bootclasspath</i> should contain all of the installed OpenFusion classes followed by the user's environment classpath. This overrides any CORBA classes defined by the JVM.

7.4.1.1 Sun Java Real-time System

The following environment variable must be set before using the *runrt* script on the Sun Java Real-Time System:

- add *<java_home>/bin* to the *PATH*.

The *runrt* script sets the default size for the scoped and immortal memory heap area using the *-XX:ScopedSize* and *-XX:ImmortalSize* flags.

7.4.1.2 IBM Websphere Real Time JVM

The following environment variables must be set before using the *runrt* script on the IBM Websphere Real Time JVM:

- Add *<JAVA_HOME>/bin* to the *PATH*.

The *runrt* script sets the default size for the scoped and immortal memory heap area using the *-Xgc:scopedMemoryMaximumSize* and *-Xgc:immortalMemorySize* flags.

The *-xrealtime* flag is used to run the Metronome real-time garbage collector and to use RTSJ services.

7.4.2 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 either:

- reading the server's IOR from a file created by the server,
- using a corbaloc URL or
- using the Naming Service.

7.5 Application Creation Example

The following example demonstrates how to create, compile, deploy and run a RTOrb-based application. This example uses the example IDL specification and source code files used for the *CORBA Hello Example*:

- The IDL is located in `<OFJ_DIR>/examples/idl/hello.idl`
- The source code is in `<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/hello`

i **Note:** The OFJ example classes are supplied precompiled, for convenience, in `ofj.jar`. Also, for convenience, the RTOrb distribution includes a very simple `Ant build.xml` script that will compile java sources located in `<OFJ_DIR>/examples`. This is an example build script that can be used and extended by developers.

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

The following IDL code declares the *GreetingService* interface in a file called `hello.idl`.

```
module com
{
  module prismtech
  {
    module ofj
    {
      module examples
      {
        module corba
        {
          interface GreetingService
          {
            string greeting(in string greetstr);
          };
        };
      };
    };
  };
};
```

An example `hello.idl` file is located in `<OFJ_DIR>/examples/idl/hello.idl`.

Step 2: Compile the application’s IDL specification with the IDL compiler to create the Java source files and classes for the stubs, skeletons and/or tie classes and interfaces.

i

Recall that the compiler is run using the `idl` script located in the `<OFJ_DIR>/bin` directory.

Although no command line options are needed for compilation, since the example uses all of the compiler’s default settings, the `-d` option is used here to specify where the generated output files should be placed. For example, both the client skeleton and server stub files are needed, default output file names are used and default file extensions are used, and the generated output is placed under the `~/myOfj/src` directory.

UNIX

```
% idl -d ~/myOfj/src hello.idl
```

Step 3: Write an implementation for the interface generated from the IDL specification.

The example implementation file is called `GreetingServiceImpl.java` and implements `class GreetingServiceImpl`. The `GreetingServiceImpl.java` file is located in `<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/hello` for the purposes of this example.

`GreetingServiceImpl` is the *servant* that will be used by the application’s server component. Note that this class should extend the POA class (generated from the IDL):

```
public class GreetingServiceImpl extends
    GreetingServicePOA
```

i

The “*Impl*” part of the `GreetingServiceImpl` name is a convention which signifies that the file is an implementation of the IDL interface specification.

Step 4: Write a server (if not being implemented by third parties). The server must:

- import the application’s implementation class (e.g. `GreetingServiceImpl`)
- have a `main()` method which instantiates the server
- have a mechanism for publishing the servant’s IOR (for object resolution by the client)

The example client implementation file is `Server.java`.

Step 5: Write a client (if not being implemented by third parties). The client must:

- import the application’s generated interface definition and associated helper class (e.g. `GreetingService` and `GreetingServiceHelper`, respectively)
- contain a `main()` method which instantiates the server

- contain a mechanism for obtaining the servant's IOR (for object resolution of the server's servant)

Step 6: Compile the developer-written source code and IDL-generated source code with a Java compiler for the platform(s) the application's components will run on.



The OFJ example classes are supplied precompiled, for convenience, within the *ofj.jar*.



Before compiling ensure that all required environment variables, RTOrb properties and other configuration settings are correctly set for the platform and RTOrb type (non real-time or real-time) the application will use (see Chapter 2, *Configuration*)

Example Compiling on Red Hat Enterprise Linux with Websphere Real Time JVM

A *non real-time* compilation on Red Hat Enterprise Linux using the *javac* compiler would use:

```
% javac -endorseddirs $OFJ_DIR ~/myOfj/src/*.java -d
~/myOfj/out/Hello
```

where

-d ~/myOfj/out/Hello specifies the output directory for the compiled class files
~/myOfj/src/.java* are the Java source files to compile (for simplicity, all Java source files are copied to the *~/myOfj/src* directory in this example)

A *real-time* compilation on Red Hat Enterprise Linux using the *javac* compiler would use:

```
% javac -endorseddirs $OFJ_DIR -Xrealtime-d ~/myOfj/out/Hello
~/myOfj/src/*.java
```

Example Compiling on Solaris with the Sun Java Real-Time System

A non real-time compilation on Solaris using the *javac* compiler would use:

```
% javac ~/myOfj/src/*.java -d ~/myOfj/out/Hello
```

where

/opt/myOFJ is the root directory where RTOrb is installed
-d ~/myOfj/out/Hello specifies the output directory for the compiled class files
~/myOfj/src/.java* are the Java source files to compile (for simplicity, all Java source files are copied to the *~/myOfj/src* directory in this example)

Step 7: Deploy the application's server and client components on the platform(s).

Copy the compiled class files and directories to the destination location where they are intended to be run from.

Step 8: Start the server and run the client.

Servers and clients:

- are usually run from different shells, like the Hello example used here
- depend on environment and configuration settings (see Chapter 2, *Configuration and RTOrb Run Scripts* on page 82.)
- may be able to use RTOrb's *run* and *runrt* convenience scripts (since they perform many or all of the configuration tasks need to run the components)¹

Example Starting the Hello Server and Client components

After changing to the directory where the Server and Client class files are located, it is recommended that the Server and Client programs are run in different windows so the output of each program can be seen separately.

Use:

```
% run com.prismtech.ofj.examples.corba.hello.Server
```

```
% run com.prismtech.ofj.examples.corba.hello.Client
```

If the call to the Server is successful, then it will return:

```
GreetingService called by Client hello Client
```

7.6 Running OpenFusion CORBA Services

PrismTech's OpenFusion CORBA Services can be used with RTOrb. RTOrb includes the OpenFusion Naming Service. Refer to the System Guide for information on running the OpenFusion CORBA Services.

1. The *run* and *runrt* scripts can be a useful source of information for running application components and creating custom run scripts.

8

Creating Applications

8.1 General

The information provided in the previous section, *Using the ORB*, describes the procedures for compiling, running and deploying applications using the ORB. This section describes how to write the applications themselves and covers:

- How to write a simple *non real-time* application, called *Hello*. This application contains the minimal, essential elements needed to create a distributed client-server application
- How to write a simple *soft real-time* version of the *Hello* application. This application demonstrates basic soft real-time programming using RTOrb.
- How to write a simple *hard real-time* version of the *Hello* application. This application demonstrates basic hard real-time programming using RTOrb.

i

It is assumed that readers understand basic CORBA programming with Java concepts and practice. The descriptions given here concentrate on those aspects which may be of most help, with basic operations (which readers should be familiar with) being only lightly covered.

8.2 A Simple Non Real-Time Application

This example, the *Hello* application, is very simple: it contains the minimum elements needed to create a working client-server application using RTOrb. The *Hello* example application is also used in Section 7.5, *Application Creation Example* to demonstrate the steps needed to compile, run and deploy applications.

Hello:

- has an IDL specification in *hello.idl* which
 - declares the *GreetingService* interface and *greeting()* function
- has a server which
 - performs the basic initialisation tasks required by all servers
 - creates a *GreetingService* servant object; the servant's single method prints a greeting for the client which called the server.
 - makes the *GreetingService* servant accessible to clients by saving the servant's stringified IOR to a file

- listens for requests from clients
- has a client which
 - performs the basic initialisation tasks required by all clients
 - obtains references to the `GreetingService` servant object by reading its stringified IOR from a file
 - calls the `greeting()` method on the `GreetingService` object which displays a greeting with the client's name



This example uses files for object resolution. Other methods of object resolution must be used on platforms which do not have a file system.

The complete source code for the Hello application is in the following RTOrb distribution directories:

```
<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/
  productguide/enterprise
<OFJ_DIR>/examples/idl
GreetingServiceImpl.java is in
  <OFJ_DIR>/examples/java/com/prismtech/ofj/examples/
```

8.2.1 IDL Specification

The IDL specification for Hello is very simple: it declares a single interface, `GreetingService`, with a single method, `greeting()`. The `greeting()` method takes a string (the name of the client calling the method) and returns a string (a greeting with the client's name).

```
module com
{
  module prismtech
  {
    module ofj
    {
      module examples
      {
        module corba
        {
          interface GreetingService
          {
            string greeting (in string greetstr);
          };
        };
      };
    };
  };
};
```



The nested module declarations, although not strictly required for this simple application, demonstrate the application's complete namespace hierarchy.

8.2.2 Java Implementation

The IDL specification for `GreetingService` is implemented in Java as *class* `GreetingServiceImpl`. This class is written by the developer. The class extends the IDL generated `GreetingServicePOA` interface.



The name of an implementation class, by convention, is derived by taking the IDL declared interface name and appending it with `Impl`.

The `GreetingServiceImpl`'s `greeting()` method is implemented simply as:

```
public String greeting (String s)
{
    System.out.println
        ("GreetingService called by Client " + s);

    String replyMsg = "Hello" + s;

    return replyMsg;
}
```

The `greeting()` method, as mentioned previously, takes a string (client's name) and returns a greeting with the name (a string).

8.2.3 Server-side

The Hello application's server component, `Server`, instantiates and activates the `GreetingService` servant. The servant's IOR is published (making it available to clients) by saving the IOR to a file. The server code is implemented in `Server.java` and is described below.

The code imports `org.omg.PortableServer.POA` and `org.omg.PortableServer.POAHelper` in order to be able to manage its servant (`GreetingService`). The server also imports the developer-written `GreetingService` implementation, `GreetingServiceImpl`.

```
import java.io.IOException;
import java.io.FileWriter;
import java.io.PrintWriter;

import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import com.prismtech.ofj.examples.corba.GreetingServiceImpl;
```

The server code declares and defines *class* `Server`.

`Server`:

- defines a `main()` method which does most of the work, such as performing initialisation tasks, POA activation and running the ORB's `run()` method (which listens for client requests)

- defines a utility method, `writeIOR()`, which publishes the servant's IOR as a stringified IOR to a file.

Initialisation and all other tasks are performed in `Server.java`'s `main()`. The following code fragment from `main()`:

- initialises the ORB and POA (using `org.omg.CORBA.ORB.init()`)
- declares, initialises and activates the `GreetingService` servant
- publishes the servant's IOR to a file which clients can use to locate the servant
- starts an event loop, the ORB's `run()` method, which waits for client requests.

```
public static void main (String[] args)
{
    try
    {
        // Initialize the ORB
        org.omg.CORBA.ORB m_orb =
            org.omg.CORBA.ORB.init (args, null);

        // Acquire Root POA
        POA rootPOA = POAHelper.narrow (
            m_orb.resolve_initial_references ("RootPOA"));

        // Create the servant
        GreetingServiceImpl gs = new GreetingServiceImpl();

        rootPOA.the_POAManager().activate();

        // Get a reference to the servant to enable clients
        // to connect to it
        org.omg.CORBA.Object obj =
            rootPOA.servant_to_reference (gs);

        // Save the servant's IOR to a file to enable clients
        // to retrieve it
        writeIOR (m_orb, obj, "hello.ior");

        System.out.println(
            "GreetingServer running... awaiting calls");

        // start a thread to listen for client requests
        m_orb.run ();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

8.2.4 Client-side

The client component of the Hello application makes requests of the server to perform tasks. The client must:

- perform basic initialisation

- obtain references to the server's `GreetingService` object
- call the `GreetingService` object's operations to perform the desired task

The client component, `Client.java`, must perform many of the same, basic initialisation tasks that the server must perform, including:

- declare ORB and `GreetingService` variables
- initialise the ORB

Items to Note

- `Client.java` imports `GreetingService` and `GreetingServiceHelper` instead of the `POA`, `POAHelper` and `GreetingServiceImpl` classes imported by the server, `Server.java`.
- The examples assume that the JVM has been started with the `ORBClass` and `ORBSingletonClass` properties using RTOrb's the supplied scripts. Accordingly, the examples do not pass them to the ORB initialisation parameters.
- The client reads the `GreetingService` servant's stringified IOR from the file previously published by the server then converts it to an object reference using the `GreetingServiceHelper.narrow()` and ORB's `narrow()` methods:

```
GreetingService gsref = GreetingServiceHelper.narrow (
                                orb.string_to_object (ior));
```

The `Client.java` source code is shown below.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.FileInputStream;
import java.io.BufferedReader;

import com.prismtech.ofj.examples.corba.GreetingService;
import com.prismtech.ofj.examples.corba.GreetingServiceHelper;

public class Client
{
    private Client ()
    {
    }

    public static void main (String[] args)
    {
        try
        {
            org.omg.CORBA.ORB orb =
                org.omg.CORBA.ORB.init (args, null);

            String ior = readIOR ("hello.ior");
            GreetingService gsref =
                GreetingServiceHelper.narrow (
                    orb.string_to_object (ior));
```

```

        if (gsref == null)
        {
            System.out.println (
                "Unable to narrow server: " + ior);
            System.exit (1);
        }

        // call GreetingService.greeting() method
        System.out.println ("Response from server is "
            + gsref.greeting(" hello Client"));
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

// Utility for reading a stringified IOR from a file
private static String readIOR (String iorFile)
    throws IOException
{
    String ior = null;
    InputStream iorURL = new FileInputStream (iorFile);
    BufferedReader in = new BufferedReader (
        new InputStreamReader (iorURL));

    ior = in.readLine();
    in.close();
    return ior;
}
}

```

8.3 A Simple Soft Real-time Application

This example, the *Soft Real-time Hello* application, is, similar to the non real-time Hello example shown above. It contains the minimum elements needed to create a working client-server application using RTOrb, but as a *soft real-time* application.

Soft Real-time Hello:

- uses *RealtimeThreads* in *HeapMemory*, threadpool and priority lane for processing client requests
- uses the same
 - IDL specification (*hello.idl*): see *IDL Specification* on page 90
 - GreetingService implementation (*GreetingServiceImpl.java*): see *Java Implementation* on page 91
 - servant object resolution technique (using a file for publishing the servant's IOR)

that was used by the Hello example

- performs the initialisation tasks specifically required for soft real-time execution in addition to the same basic initialisation tasks performed by Hello

The soft real-time associated tasks and procedures are described below. The complete source code for the Real-time Hello application is located in:

```
<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/productguide/soft
  <OFJ_DIR>/examples/idl/
  GreetingServiceImpl.java is in
  <OFJ_DIR>/examples/java/com/prismtech/ofj/examples.
```

8.3.1 Server-side

The Soft Real-time Hello application's server component, *Server*, instantiates and activates the *GreetingService* servant, similar to the Hello application, but performs additional initialisation tasks needed for real-time operation.

The server code is implemented in the *Server.java* file located in the `<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/productguide/soft` directory.

Like Hello, the soft real-time *Server.java* code imports *org.omg.PortableServer.POA*, *org.omg.PortableServer.POAHelper* and the *GreetingService* implementation, *GreetingServiceImpl*. However, the code also imports classes which are needed for soft real-time threading, memory allocation and memory management. These additional classes are shown in **bold** in the following code fragment.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.io.FileWriter;

import javax.realtime.RealtimeThread;

import org.omg.PortableServer.ImplicitActivationPolicyValue;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import com.prismtech.ofj.examples.corba.GreetingServiceImpl;
```

The server code declares and defines *class Server*. *Server* extends *RealtimeThread*. Classes which run *RealtimeThread* threads must inherit *RealtimeThread*. *Server* has an empty, default constructor.

```
public class Server extends RealtimeThread
{
    // Variable for holding arguments passed to the ORB
    private String[] m_args;

    public Server ()
    {
    }
}
```

The class' *main()* method performs some basic initialisation tasks (the remaining initialisation tasks have been delegated to the class' *run()* method) and starts a real-time thread which, indirectly, runs the ORB's *run()* method (which listens for client requests).

The first initialisation tasks that *main()* performs is to allocate a server instance to heap memory, pass any command line arguments to the server instance and start the server's real-time thread.

```
public static void main (String[] args)
{
    try
    {
        // Allocate server instance to heap memory
        Server srv = new Server ();
        MemoryArea mem = ImmortalMemory.instance();

        // Pass any command line arguments to this new instance
        srv.setargs(args);
    }
}
```

The server's real-time thread can be started after these initialisation tasks are complete.

```
// start real-time thread
srv.start();
```

Note that class *Server* has inherited the *start()* method from class *RealtimeThread*. *RealtimeThread.start()* calls the inherited class' *run()* method, in this case, *Server.run()*.

Server's run() method performs tasks which initialise the ORB, acquire the real-time ORB and POA, as well as doing the other tasks that are needed for making the *GreetingService* servant available for processing requests from clients.

```
// overrides RealtimeThread::run(), called by srv.start()
public void run()
{
    try
    {
        // Initialize the ORB and real-time ORB. Must be called
        // from RealtimeThread::run() or exception is thrown
        org.omg.CORBA.ORB m_orb =
            org.omg.CORBA.ORB.init (args, null);

        // Acquire Root POA
        POA rootPOA =
            POAHelper.narrow(
                m_orb.resolve_initial_references("RootPOA"));

        // Acquire RTORB
        org.omg.RTCORBA.RTORB rtORB =
            org.omg.RTCORBA.RTORBHelper.narrow (
                m_orb.resolve_initial_references("RTORB"));
    }
}
```

The first set of tasks, shown in the code fragment above, initialises the ORB and obtains references to the POA and the *real-time* ORB.

The next set of tasks that `run()` performs is to establish a threadpool and priority threadpool lane that the `GreetingService` servant can use for processing, since this is a real-time server. Only one priority lane is created in this example, since the example does not do very much. However, more powerful applications which anticipate multiple, simultaneous client requests would likely use more than one threadpool lane.

The comments shown in the code below describe the properties, being passed as arguments, that are used by the `ThreadpoolLane` constructor and `rtORB.create_threadpool_with_lanes()` method to configure the threadpool and lane, respectively.



Threadpool lanes must be created before the threadpool since they are passed to `create_threadpool_with_lanes()` as elements of an array of lanes.

```
// Initialise a threadpool and one priority lane which
// the GreetingService servant (GreetingServiceImpl)
// will use for processing requests
org.omg.RTCORBA.ThreadpoolLane[] lanes
    = new org.omg.RTCORBA.ThreadpoolLane[1];

// Create priority lane
lanes[0] = new org.omg.RTCORBA.ThreadpoolLane
(
    (short) 15000, // Default CORBA Priority assigned
    1,           // Number of static threads in lane
    0           // Number of dynamic threads
);

// Create a threadpool for the ThreadpoolLane
int pool_1 = rtORB.create_threadpool_with_lanes
(
    32 * 1024, // stack size for threads in pool
    lanes,    // the lanes
    true,     // allow borrowing threads between pools
    true,     // allow request buffering
    1000,    // max number buffered requests allowed
    1000000  // max size of request_buffer (Bytes)
);
```

The servant will be associated with a real-time POA, *RTPOA*, that has been configured with appropriate policies. The real-time POA in this example has a thread policy with a single threadpool, a client propagated priority model and an implicit activation policy.

The following code fragment shows how the POA policies are set, real-time POA created and activated, and servant instance associated with the POA.

```
// create policies for the RTPOA that will be
// associated with the Hello servant, RTThreadpool
// and the Lanes

// threadpool policy for RTPOA
org.omg.RTCORBA.ThreadpoolPolicy tp_policy =
    rtORB.create_threadpool_policy (pool_1);

// priority model for the real-time execution policy
org.omg.RTCORBA.PriorityModelPolicy pm_policy =
    rtORB.create_priority_model_policy
    (
        org.omg.RTCORBA.PriorityModel.CLIENT_PROPAGATED,
        (short) 1
    );

// implicit activation policy
org.omg.CORBA.Policy ia_policy =
    rootPOA.create_implicit_activation_policy (
        ImplicitActivationPolicyValue.IMPLICIT_ACTIVATION);

// CORBA Policies for instantiating the RTPOA
org.omg.CORBA.Policy[] policies =
{
    tp_policy, // threadpool policy
    pm_policy, // priority model policy
    ia_policy, // implicit activation policy
};

// Create POA using policies defined above
// Acquire POAManager
org.omg.PortableServer.POAManager manager =
    rootPOA.the_POAManager();

// Acquire the rootPOA
org.omg.RTPortableServer.POA this_rootPOA =
    org.omg.RTPortableServer.POAHelper.narrow (rootPOA);

// Create RTPOA on the rootPOA using policies defined above
POA my_RTPOA =
    this_rootPOA.create_POA("myRTPOA", manager, policies);

// Create the servant
GreetingServiceImpl gs = new GreetingServiceImpl();

// Add to and activate the servant in the RTPOA
my_RTPOA.activate_object (gs);

// Get a reference to the servant (to enable clients
// to obtain its IOR)
org.omg.CORBA.Object obj =
    my_RTPOA.servant_to_reference (gs);

// Activate the RTPOA
my_RTPOA.the_POAManager().activate();
```


The stringified IOR of the servant instance is published to a file (using the Server's utility, `writeIOR()`) enabling clients to obtain the IOR and resolve the servant object. After publishing the IOR, the *ORB's* `run()` method is executed and begins listening for client requests.

```
// Publish servant's stringified IOR to a file
writeIOR (m_orb, obj, "hello.ior");

System.out.println(
    "GreetingServer running in RT... awaiting calls");

// start thread to listen for client requests
m_orb.run ();
}
catch (Exception e)
{
    e.printStackTrace();
}
}
```

The following code fragment shows the Server's `writeIOR()` and `setargs()` utility methods.

```
// Utility for writing stringified IOR to a file
private static void writeIOR
(
    org.omg.CORBA.ORB orb,
    org.omg.CORBA.Object objref,
    String filename
)
{
    try
    {
        PrintWriter pw = new PrintWriter (
            new FileWriter (filename));
        pw.println (orb.object_to_string(objref));
        pw.flush();
        pw.close();
    }
    catch (IOException ioe)
    {
        System.out.println (
            "Encountered exception writing " + filename);
        System.exit (0);
    }
}

// Utility for setting application arguments
public void setargs (String[] args)
{
    m_args = args;
}
```

8.3.2 Client-side

The Soft Real-time Hello application's client component, *Client*, performs the same basic initialisation tasks that its non real-time Hello application counterpart did, but it also performs additional initialisation tasks which are needed for soft real-time operation and similarly as needed by the soft real-time *Server* component.

The client code is implemented in the *Client.java* file located in the `<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/productguide/soft` directory.

The soft real-time *Client* code imports the same classes as the non real-time version, although it also imports the classes which are needed for soft real-time operation. These additional classes are shown in **bold** in the code fragment below.

```
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.IOException;

import javax.realtime.RealtimeThread;

import org.omg.RTCORBA.CurrentHelper;

import com.prismtech.ofj.examples.corba.GreetingService;
import com.prismtech.ofj.examples.corba.GreetingServiceHelper;
```

The client code declares and defines *class Client*. *Client* extends *RealtimeThread*, the same as class *Server* in the server component.

```
public class Client extends RealtimeThread
{
    // variable to hold arguments
    private String[] args;

    org.omg.CORBA.ORB orb;

    // Variable to hold reference to GreetingService server
    public static GreetingService gsref;

    public Client()
    {
    }
}
```

The *main()* method performs some basic initialisation tasks: the tasks are similar to those performed by class *Server*.

The first task is to initialise the client object and allocate it to heap memory. The *clnt.setargs(args)* call, shown in last line of the following code, uses the class' *setargs()* utility method to forward any command line arguments passed to the client on to the ORB when it is initialised. The client instance is then run using the class' *start()* method.

```

public static void main (String[] args)
{
    Client clnt = new Client ();

    clnt.setargs (args);

    // Starting the real-time thread
    clnt.start();
}

```

Recall that `start()` is inherited from `RealtimeThread` and that it calls the class' own `run()` method, which:

- initialises the ORB
- retrieves the IOR for the Server's `GreetingService` servant (using the Client's `readIOR()` utility method)
- obtains a reference to the servant (using `GreetingServiceHelper.narrow()`)
- makes a request on the servant (`gsref.greeting(" rthello Client")`)

```

public void run()
{
    try
    {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args, null);
        String ior = readIOR ("hello.ior");
        gsref = GreetingServiceHelper.narrow (
            orb.string_to_object (ior));

        if (gsref == null)
        {
            System.out.println ("Unable to narrow server: " + ior);
            System.exit (1);
        }

        //get and set the priority with RTCORBA::Current variable
        org.omg.RTCORBA.Current rtc;
        tc = CurrentHelper.narrow (
            orb.resolve_initial_references ("RTCCurrent"));

        // Set local current thread to low priority in the client
        tc.the_priority((short) 15000);

        // Call (the remote) GreetingService's greeting method
        System.out.println ("Response from server is "
            gsref.greeting(" rthello Client"));
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

If the call to the Server is successful, then its servant, `GreetingService`, will return:

`GreetingService` called by Client `rthello` Client

8.4 A Simple Hard Real-time Application

This example, the *Hard Real-time Hello* application, is, similar to the non real-time Hello example shown above. It contains the minimum elements needed to create a working client-server application using RTOrb, but as a *hard real-time* application.

Hard Real-time Hello:

- uses `NoHeapRealtimeThreads` in `ImmortalMemory`, threadpool and priority lane for processing client requests
- uses the same
 - IDL specification (`hello.idl`): see *IDL Specification* on page 90
 - `GreetingService` implementation (`GreetingServiceImpl.java`): see *Java Implementation* on page 91
 - servant object resolution technique (using a file for publishing the servant's IOR)

that was used by the Hello example

- performs the initialisation tasks specifically required for hard real-time execution in addition to the same basic initialisation tasks performed by Hello

The hard real-time associated tasks and procedures are described below. The complete source code for the Real-time Hello application is located in:

```
<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/productguide/hard
```

```
<OFJ_DIR>/examples/idl
```

```
GreetingServiceImpl.java is in
```

```
<OFJ_DIR>/examples/java/com/prismtech/ofj/examples.
```

8.4.1 Server-side

The Hard Real-time Hello application's server component, `Server`, instantiates and activates the `GreetingService` servant, similar to the Hello application, but performs additional initialisation tasks needed for hard real-time operation.

The server code is implemented in the `Server.java` file located in the `<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/productguide/hard` directory.

Like Hello, the hard real-time `Server.java` code imports `org.omg.PortableServer.POA`, `org.omg.PortableServer.POAHelper` and the `GreetingService` implementation, `GreetingServiceImpl`. However, the code also imports classes which are needed for hard real-time threading, memory allocation and memory management. These additional classes are shown in **bold** in the following code fragment.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.io.FileWriter;

import javax.realtime.ImmortalMemory;
import javax.realtime.NoHeapRealtimeThread;
import javax.realtime.RealtimeThread;

import org.omg.PortableServer.ImplicitActivationPolicyValue;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import com.prismtech.ofj.examples.corba.GreetingServiceImpl;
```

The server code declares and defines `class Server`. `Server` extends `NoHeapRealtimeThread`. Classes which run `RealtimeThread` threads must inherit `NoHeapRealtimeThread`. The server's constructor calls the `NoHeapRealtimeThread` superclass, passing a pointer to the singleton `ImmortalMemory` object.

```
public class Server extends NoHeapRealtimeThread
{
    // Variable for holding arguments passed to the ORB
    private static String[] args;

    public Server ()
    {
        super (null, ImmortalMemory.instance());
    }
}
```

The class' `main()` method performs some basic initialisation tasks (the remaining initialisation tasks have been delegated to the class' `run()` method) and starts a real-time thread, `NoHeapRealtimeThread`, which indirectly runs the ORB's `run()` method (which listens for client requests).

The first initialisation tasks that `main()` performs is to allocate a server instance to immortal memory, pass any command line arguments to the server instance and start the server's real-time thread. Using immortal memory allows the server to be available to all of the application's threads the entire time its Java runtime environment is running (see Section 6.1.2, *Memory Management*, on page 72).

```
public static void main (String[] args)
{
```

```

try
{
    // instantiate to immortal memory using class loader
    Server srv = Server) ImmortalMemory.instance ().newInstance
    (Class.forName(
    "com.prismtech.ofj.examples.corba.productguide.hard.Server"));

    // Pass any command line arguments to this new instance
    srv.setargs(args);
}

```

The server's real-time thread, `NoHeapRealtimeThread`, can be started after these initialisation tasks are complete *and* if the real-time JVM is able to schedule the thread to run.

```

// Check that the JVM can schedule the server thread to run
if (!srv.getScheduler().isFeasible())
{
    System.out.println(
        "RTCORBA HelloServer cannot be scheduled to run.");
}
else
{
    // start real-time thread
    srv.start();
}

```

Note that class `Server` has inherited the `start()` method from class `NoHeapRealtimeThread`. `NoHeapRealtimeThread.start()` calls the inherited class' `run()` method, in this case, `Server.run()`.

`Server's run()` method performs tasks which initialise the ORB, acquire the real-time ORB and POA, as well as doing the other tasks that are needed for making the `GreetingService` servant available for processing requests from clients.

```

// overrides RealtimeThread::run(), called by srv.start()
public void run()
{
    try
    {
        // Initialize the ORB and real-time ORB. Must be called
        // from RealtimeThread::run() or exception is thrown
        org.omg.CORBA.ORB m_orb =
            org.omg.CORBA.ORB.init (args, null);

        // Acquire Root POA
        POA rootPOA =
            POAHelper.narrow(
                m_orb.resolve_initial_references("RootPOA"));

        // Acquire RTORB
        org.omg.RTCORBA.RTORB rtORB =
            org.omg.RTCORBA.RTORBHelper.narrow (
                m_orb.resolve_initial_references("RTORB"));
    }
}

```

The first set of tasks, shown in the code fragment above, initialises the ORB and obtains references to the POA and the *real-time* ORB.

The next set of tasks that `run()` performs is to establish a threadpool and priority threadpool lane that the `GreetingService` servant can use for processing, since this is a real-time server. Only one priority lane is created in this example, since the example does not do very much. However, more powerful applications which anticipate multiple, simultaneous client requests would likely use more than one threadpool lane.

The comments shown in the code below describe the properties, being passed as arguments, that are used by the `ThreadpoolLane` constructor and `rtORB.create_threadpool_with_lanes()` method to configure the threadpool and lane, respectively.



Threadpool lanes must be created before the threadpool since they are passed to `create_threadpool_with_lanes()` as elements of an array of lanes.

```
// Initialise a threadpool and one priority lane which
// will be used for processing requests
org.omg.RTCORBA.ThreadpoolLane[] lanes
    = new org.omg.RTCORBA.ThreadpoolLane[1];

// Create priority lane
lanes[0] = new org.omg.RTCORBA.ThreadpoolLane
(
    (short) 15000, // Default CORBA Priority assigned
    1,           // Number of static threads in lane
    0           // Number of dynamic threads
);

// Create a threadpool for the ThreadpoolLane
int pool_1 = rtORB.create_threadpool_with_lanes
(
    32 * 1024, // stack size for threads in pool
    lanes,    // the lanes
    true,     // allow borrowing threads between pools
    true,     // allow request buffering
    1000,     // max number buffered requests allowed
    1000000   // max size of request_buffer (Bytes)
);
```

The servant will be associated with a real-time POA, *RTPOA*, that has been configured with appropriate policies. The real-time POA in this example has a thread policy with a single threadpool, a client propagated priority model and an implicit activation policy.

The following code fragment shows how the POA policies are set, real-time POA created and activated, and servant instance associated with the POA.

```
// create policies for the RTPOA that will be
// associated with the Hello servant, RTThreadpool
// and the Lanes
```

```

// threadpool policy for RTPOA
org.omg.RTCORBA.ThreadpoolPolicy tp_policy =
    rtORB.create_threadpool_policy (pool_1);

// priority model for the real-time execution policy
org.omg.RTCORBA.PriorityModelPolicy pm_policy =
    rtORB.create_priority_model_policy
    (
        org.omg.RTCORBA.PriorityModel.CLIENT_PROPAGATED,
        (short) 15000
    );

// implicit activation policy
org.omg.CORBA.Policy ia_policy =
    rootPOA.create_implicit_activation_policy (
        ImplicitActivationPolicyValue.IMPLICIT_ACTIVATION);

// CORBA Policies for instantiating the RTPOA
org.omg.CORBA.Policy[] policies =
{
    tp_policy, // threadpool policy
    pm_policy, // priority model policy
    ia_policy, // implicit activation policy
};

// create POA using policies defined above
// acquire POAManager
org.omg.PortableServer.POAManager manager =
    rootPOA.the_POAManager();

// acquire the rootPOA
org.omg.RTPortableServer.POA this_rootPOA =
    org.omg.RTPortableServer.POAHelper.narrow (rootPOA);

// create RTPOA on the rootPOA using policies defined above
POA my_RTPOA =
    this_rootPOA.create_POA("myRTPOA", manager, policies);

// create the servant
GreetingServiceImpl gs = new GreetingServiceImpl();

// add to and activate the servant in the RTPOA
my_RTPOA.activate_object (gs);

// Get a reference to the servant (to enable clients
// to obtain its IOR)
org.omg.CORBA.Object obj =
    my_RTPOA.servant_to_reference (gs);

// Activate the RTPOA
my_RTPOA.the_POAManager().activate();

```

The stringified IOR of the servant instance is published to a file (using the Server's utility, `writeIOR()`) enabling clients to obtain the IOR and resolve the servant object. After publishing the IOR, the `ORB`'s `run()` method is executed and begins listening for client requests.

```

// Publish servant's stringified IOR to a file
writeIOR (m_orb, obj, "hello.ior");

```



```

        System.out.println(
            "GreetingServer running in RT... awaiting calls");

        // start thread to listen for client requests
        m_orb.run ();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

The following code fragment shows the Server's *writeIOR()* and *setargs()* utility methods.

writeIOR()

```

// Utility for writing stringified IOR to a file
private static void writeIOR (
    org.omg.CORBA.ORB orb,
    org.omg.CORBA.Object objref,
    String filename)
{
    try
    {
        PrintWriter pw = new PrintWriter (
            new FileWriter (filename));
        pw.println (orb.object_to_string(objref));
        pw.flush();
        pw.close();
    }
    catch (IOException ioe)
    {
        System.out.println (
            "Encountered exception writing " + filename);
        System.exit (0);
    }
}

// Utility for setting application arguments
public void setargs (String[] args)
{
    m_args = args;
}

```

setargs()



The input arguments to *setargs()* must be copied to immortal memory when using hard real-time mode.

```

// Utility for setting application arguments
public void setargs (final String [] in_args)
{
    RealtimeThread allocator = new RealtimeThread (
        (null, null, null, ImmortalMemory.instance ()),

```

```

        null, new Runnable ()
        {
            public void run ()
            {
                String[] oldArgs = in_args;
                String[] newArgs = new String[ oldArgs.length ];
                for (int i = 0; i < oldArgs.length; i++)
                {
                    newArgs[ i ] = oldArgs[ i ];
                }

                args = newArgs;
            }
        });

allocator.start ();

try
{
    allocator.join ();
}
catch (InterruptedException ie)
{
}
}

```

8.4.2 Client-side

The Hard Real-time Hello application's client component, *Client*, performs the same basic initialisation tasks that its non real-time Hello application counterpart did, but it also performs additional initialisation tasks which are needed for hard real-time operation and similarly as needed by the hard real-time Server component.

The client code is implemented in the *Client.java* file located in the `<OFJ_DIR>/examples/java/com/prismtech/ofj/examples/corba/prod uctguide/hard` directory.

The hard real-time Client code imports the same classes as the non real-time version, although it also imports the classes which are needed for hard real-time operation. These additional classes are shown in **bold** in the code fragment below.

```

import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.IOException;

import javax.realtime.ImmortalMemory;
import javax.realtime.NoHeapRealtimeThread;
import javax.realtime.RealtimeThread;

import org.omg.RTCORBA.CurrentHelper;

import com.prismtech.ofj.examples.corba.GreetingService;
import com.prismtech.ofj.examples.corba.GreetingServiceHelper;

```

The client code declares and defines *class Client*. *Client* extends *NoHeapRealtimeThread*, the same as class *Server* in the server component.

```
public class Client extends NoHeapRealtimeThread
{
    // variable to hold arguments
    private String[] args;

    org.omg.CORBA.ORB orb;

    // Variable to hold reference to GreetingService server
    public static GreetingService gsref;

    public Client()
    {
        super (null, ImmortalMemory.instance ());
    }
}
```

The *main()* method performs some basic initialisation tasks and checks to see it can run in the real-time JVM: the tasks are similar to those performed by class *Server*.

The first task is to initialise the client object and allocate it to immortal memory. The *clnt.setargs (args)* call, shown in last line of the following code fragment, uses the class' *setargs()* utility method to forward any command line arguments passed to the client on to the ORB when it is initialised.

```
public static void main (String[] args)
{
    Client clnt = null;

    try
    {
        clnt = (Client) ImmortalMemory.instance ().newInstance
            (Class.forName (
                "com.prismtech.ofj.examples.corba.productguide.hard.Client"));
    }
    catch (Exception e)
    {
        System.out.println ("exc caught" + e.toString());
    }

    clnt.setargs (args);
}
```

The last two tasks that *main()* does, which are similar to the final tasks performed by the *Server.main()* method, are to determine if the client instance can be scheduled to be run by the JVM and if it can, then call the class' *start()* method.

```
if (!clnt.getScheduler().isFeasible())
{
    // not possible to run at this time
    System.out.println(
        "Running RTCORBA GreetingService Client is not feasible");
}
```

```

else
{
    // can schedule to run
    clnt.start();
}

```

The client instance is then run using the class' `start()` method. Recall that `start()` is inherited from `NoHeapRealtimeThread` and that it calls the class' own `run()` method.

If the client is able to be scheduled to run by the JVM, then `run()` is called and it

- initialises the ORB
- retrieves the IOR for the Server's `GreetingService` servant (using the Client's `readIOR()` utility method)
- obtains a reference to the servant (using `GreetingServiceHelper.narrow()`)
- makes a request on the servant (`gsref.greeting (" rthello Client")`)

```

public void run()
{
    try
    {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args, null);
        String ior = readIOR ("hello.ior");
        gsref = GreetingServiceHelper.narrow (
            orb.string_to_object (ior));

        if (gsref == null)
        {
            System.out.println ("Unable to narrow server : " + ior);
            System.exit (1);
        }

        // get and set the priority using RTCORBA::Current
        org.omg.RTCORBA.Current rtc;
        rtc = CurrentHelper.narrow
            (orb.resolve_initial_references ("RTCCurrent"));

        // set local current thread to low priority in client
        rtc.the_priority((short) 15000);

        // remotely call GreetingService's greeting method
        System.out.println ("Response from server is "
            + gsref.greeting(" rthello Client"));
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

If the call to the Server is successful, then its servant, `GreetingService`, will return:

```
GreetingService called by Client rthello Client
```


A close-up, low-angle photograph of a computer keyboard. The keys are white and the background is a dark, muted blue. A white grid pattern is overlaid on the image, creating a sense of depth and structure. The lighting is soft, highlighting the texture of the keys and the grid lines.

APPENDICES

Appendix

A API Enhancements

RTOrb provides enhancements to the OMG's standard CORBA API. These enhancements described below

Classes and Methods

InputStream Class

```
com.prismtech.ofj.orb.InputStream
```

InputStream ()

```
public InputStream (org.omg.CORBA.ORB, byte[])
```

This is the *InputStream* class constructor for creating *InputStreams*.

reset ()

```
public void reset ()
```

reset resets the stream to a pristine initial state. This should only be used with non-allocator based objects. It will not remove or erase the internal contained byte array. It will only reset the read and write pointers.

setBuffer ()

```
public void setBuffer (byte[])
```

setBuffer replaces the internal buffer with the supplied byte array. This also implicitly resets the stream.

setLittleEndian ()

```
public void setLittleEndian (boolean endian)
```

setLittleEndian() sets the ordering of bytes in the buffer to have lower significance at lower addresses or the *little end*.

OutputStream Class

```
com.prismtech.ofj.orb.OutputStream
```

Use the standard OMG *create_output_stream()* method on *org.omg.CORBA.ORB* to construct an *OutputStream*.

getBufferCopy()

```
public byte [] getBufferCopy()
```

getBufferCopy returns the marshalled data by returning a *copy* of the byte array data for this stream.

i Only a copy of the stream's *used* bytes are returned.

getWritePosition ()

```
public int getWritePosition ()
```

getWritePosition() gets the stream's current write position.

reset ()

```
public void reset ()
```

reset() resets the stream to a pristine initial state. **reset()** should only be used with non-allocator based objects. **reset()** will not remove or erase the internal contained byte array: it only resets the write and used pointers.

setBuffer ()

```
public void setBuffer (byte[])
```

setBuffer() replaces the internal buffer with the supplied byte array. **setBuffer()** also implicitly resets the stream. **setBuffer()** leaves the write position at zero.

setWritePosition ()

```
public void setWritePosition (int)
```

setWritePosition() sets the current write position of the stream. A *org.omg.CORBA.MARSHAL* exception will occur if **setWritePosition()** is set outside the stream's bounds.

setWritePosition() is intended to allow users to overwrite data that has already been written into the stream. If a value of *x* is equal to **size()**, then calling **setWritePosition()** and writing beyond *x* will produce undefined behaviour.

size ()

```
public int size ()
```

size() returns the current size of the stream (in other words, the amount of written data).

i The underlying byte buffer may be longer than **size()** or the value of *writePos*.

Valuetypes and Factories

It is possible to use a valuetype without providing a corresponding factory. This feature can be used by supplying an implementation class which has a name of the form:

```
<vt>Impl
```

where `<vt>` is the name of the valuetype defined in the IDL.

For example, if the valuetype is called *Thing*, then the implementation class must be called *ThingImpl*.

The implementation class must be on the classpath so that the ORB can find it at run time.

Examples

Example 1 Writing a valuetype to a stream

Using a valuetype called *RetrievalResult*, this examples shows how to write a valuetype to a stream.

The *RetrievalResult* is defined in IDL as:

```
valuetype RetrievalResult
{
    private FloatSequence scores;
    private WStringValueSequence ids;
    private long size;

    FloatSequence getScores();
    WStringValueSequence getIds();
    long getSize();

    void setScores(in FloatSequence scores);
    void setIds(in WStringValueSequence ids);
    void setSize(in long size);

    factory init(in FloatSequence scores,
                in WStringValueSequence ids, ins long size);
};
```

The *RetrievalResult* valuetype is first retrieved from the server. A long placeholder is written to later write the size into. Next the valuetype itself is written to the stream.

As shown in the following code example, it is possible to get a copy of the written bytes and create an inputstream for reading from those bytes.

Finally, by recording the final write position it is possible to calculate the size of the *RetrievalResult* and record its size in the stream.

```
1 RetrievalResult rresult = server.search();
2 OutputStream os = (OutputStream)orb.create_output_stream ();
3 System.err.println ("Stream size " + os.size () +
4     " and write position " + os.getWritePosition ());
```

```

5
6 int offset = os.getWritePosition ();
7 // Placeholder for length
8 os.write_long (0);
9
10 os.write_value (rresult);
11 int offset2 = os.getWritePosition ();
12 System.err.println ("Offset2 = " + offset2 + " and " +
13     os.size ());
14
15 // Test the value has been written.
16 byte []lb = os.getBufferCopy ();
17 InputStream inputStr =
18     new com.prismtech.ofj.orb.InputStream (getOrb (), lb);
19 assertTrue (0 == inputStr.read_long ());
20 assertTrue
21     (Arrays.equals (rresult.getScores (),
22         (RetrievalResultHelper.read (inputStr)).getScores ()));
23
24 // Write the actual size of the RetrievalResult
25 os.setWritePosition (offset);
26 os.write_long (offset2 - offset);
27 // Reset the write position to the original.
28 os.setWritePosition (offset2);
29
30 System.err.println ("Stream size " + os.size () +
31     " and write position " + os.getWritePosition ());

```

Example 2 Reusing an OutputStream

This example shows how to set the buffer of an existing `OutputStream`. The stream created in *Example 1* is used. Additional information is then written into the stream.

An IOR is written to a new byte buffer then, as in *Example 1*, `RetrievalResult` is written to the buffer.

However, this time a new `InputStream` is created from the outputstream and the data is read back again.

```

1 // Dummy ior string
2 bytString ior =
3 "IOR0000000000000100000000000000A0000102000000000931302E312E302E3400
4 00";
5 byte []iorB = new byte [ 4 + ior.getBytes ().length ];
6
7 // Use a new byte array with an existing stream.
8 os.setBuffer (iorB);
9
10 // Write the ior bytes out.
11 os.write_long (ior.getBytes ().length);
12 os.write (ior.getBytes());
13
14 // Write the long and RetrievalResult as before.
15 offset = os.getWritePosition ();
16 os.write_long (0);

```

```
15 os.write_value (rresult);
16
17 System.err.println ("Stream size " + os.size () +
18     " and write position " + os.getWritePosition ());
19
20 os.setWritePosition (offset);
21 os.write_long (os.size() - offset);
22 os.setWritePosition (os.size ());
23
24 // Read the data.
25 InputStream is = os.create_input_stream ();
26 int strLength = is.read_long ();
27 System.err.println ("Read string length of " + strLength);
28 byte []str = new byte[strLength];
29 is.read_octet_array (str, 0, strLength);
30 System.err.println ("Read string of " + new String (str));
31 int valueSize = is.read_long ();
32 RetrievalResultHelper.read (is);
```


A close-up, low-angle photograph of a computer keyboard. The keys are white and slightly blurred, creating a sense of depth. A white grid pattern is overlaid on the image, consisting of thin lines that intersect to form a mesh. The overall color palette is a soft, muted blue-grey. The word "BIBLIOGRAPHY" is centered in the upper half of the image in a dark blue, bold, sans-serif font.

BIBLIOGRAPHY

Bibliography

The documents and articles listed below are referred to in the text or are recommended reading.

- [1] *A Comprehensive Source of Information on Real-time Systems and Design*, Jensen D., <http://www.real-time.org>.
- [2] *Concurrent and Real-Time Programming in Java*, Andy Wellings, John Wiley & Sons Ltd., 2004.
- [3] *Patterns for Concurrent and Networked Objects*, Pattern Oriented Software Architecture - Volume 2, Schmidt D., et. al., J Wiley, 2000.
- [4] *Predictable Scheduling Algorithms and Applications*, Hard Real-time Computing Systems, Buttazo G., Kluwer Academic Press, 1997.
- [5] *Programming for the Real World*, Posix.4, Gallmeister B.O., O'Reilly and associates, 1995.
- [6] *Real-Time Java Programming*, Peter C. Dibble, Sun Microsystems Press Java Series, 2002.
- [7] *Real-Time Specification for Java (RTSJ) v1.0.1*, Rudy Belliardi, et. al., <http://www.rtsj.org>
- [8] *Real-Time Specification for Java*, Bollella G., et. al., Addison Wesley, 2000.
- [9] *Real-Time Systems and Programming Languages*, Alan Burns and Andy Wellings, Addison Wesley, Third Edition, 2001.
- [10] *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kopetz, H., Kluwer Academic Press, Fourth Edition, 1997.
- [11] *Sun Java Real-Time System*, <http://java.sun.com/j2se/realtime>, Sun Microsystems.
- [12] *Synchronization in Real-time Systems: A Priority Inheritance Approach*, Rajkumar R., Kluwer Academic Press, 1991.
- [13] *What is Predictability for Real-time Systems*, Stankovic J.A. and Ramamritham K., Journal of Real-time Systems, Issue 2, 1990.



INDEX

Index

A

abstraction, object reference	42	Asynchronous Event Handling	73
adapters, object	44	asynchronous event handling	73
Advice Notes	78	Asynchronous Thread Termination	73
Application Creation Example	84	asynchronous thread termination	73
associate		Asynchronous Transfer Of Control	73
priority	52	asynchronous transfer of control	73
priority inheritance protocol	69	avoidance techniques	55
thread pools, poa	64		

B

basic object adapter interface	44	execution times and predictability	53
Bibliography	123	priority inversion	67
BOA interface	44	system call execution times	55
bounded			

C

cache	55	Configuring OpenFusion CORBA Services	29
character strings	42	connections	
Classes and Methods	115	non-multiplexed	61
client	38, 42	control, asynchronous transfer of	73
definition	42	Conventions	9, 79
different terms	42	corba	
processing context	42	real-time mutex interface	70
role	38	corba model	
stub	38	location transparency	35
Client and Server Protocol Configuration	61	CORBA Mutex	69
client and server protocol configuration	61	corba mutex	69
Client-side	100	CORBA Priority	68
Common Requirements	81	corba priority	68
Compiling Applications	81	CORBA Priority Mapping	68
Compliance	4	corba priority mapping	68
compliance	4	corba specification	36
computing, distributed object	36	corba to native priority	68
definition	33	CorbaServices	36
Configuration of the Hard Real-Time Mode	19	Current	62
Configuration of the Soft Real-Time Mode	18	current	62
Configuration Options and Properties	15	Current interface	69
Configuration Properties	20	current interface	69

D

- delivering requests. 39, 41
 - Deploying and Running Applications. 82
 - Developing Real-time Systems with RTOS . . . 52
 - dispatching, threads. 71
 - distributed object 36
 - distributed object computing. 33, 36
 - distributed object technology 34
 - distributed systems
 - important properties 53
 - predictability 54
-

E

- Enterprise Mode 17
 - environment variables 15
 - event handling, asynchronous 73
 - Examples. 117
 - Examples Information. 74
 - execution times, bounded 53
-

F

- Features 4, 54
 - first class object, righteous 43
 - first class objects. 43
 - Further Reading. 74
-

G

- getBufferCopy() 115
 - getWritePosition () 116
-

H

- Hard Real-time Application 102
 - Hard Real-Time Mode 19
-

I

- IBM Websphere Real Time JVM 82, 83
- IDL
 - mutex 70
- idl. 37
 - mutex 70
 - rtcorba priority 68
- IDL Specification 90
- inheritance
 - mutex and resource contention 69
- InputStream () 115
- InputStream Class 115
- Install the Licence File 12
- Installation. 11
- installation 11
- Installation Procedure 10
- installing
 - command line mode 11
 - gui mode 11
- Installing Using Command Line Mode. 11
- Installing Using GUI Mode. 11
- Intended Audience xi
- interface
 - client 36
 - contract 37
 - current 69
 - definition 36
 - implementation 37
 - inheritance 37
 - mapping priorities 68
 - mutex 70
 - object implementation 36
- Interface Definition Language (IDL) 37

interface, base object adapter	44	interrupts.	55
interface, BOA	44	Introduction	77
interface, programming, orb pseudo object	43	Invocation Timeouts.	61
interrupt triggering	55	invocation timeouts	61

J

Java Compiler.	81	JVM Configuration	28
Java Implementation.	91		

L

Laned Threadpool.	64	licence file, installing	12
laned threadpool	64	location transparency	34
language mapping.	37	Logging	28

M

mapping	37	IDL	70
mapping priorities.	68	inheritance	69
mediation by orb.	39	notifies in rt corba.	69
Memory Management	18, 19, 72	priority protocol	69
memory management	72	specification requirement	69
Messaging Configuration	27	mutex interface	
method invocation	49	real-time	70
Multiple ORBs in a Single JVM	20	Mutex Notifies in RT CORBA	69
mutes.	69	Mutex, priority protocol.	69
mutex			

N

native priority and priority mappings.	58	Non-Determinism.	54
Native Priority and PriorityMappings	58	non-determinism.	54
network		Non-Multiplexed Connections	61
controlling resources.	63	non-multiplexed connections	61

O

object		target	34
adapters	44	object adapter interface	44
pseudo objects	44	object computing, distributed.	33, 36
first class	43	Object Key Map	28
pseudo	43	object reference	42
righteous	43	abstraction.	42
services, fundamental, standard interfaces	36	character strings	42

definition	42	c++ polymorphism	40
object technology, distributed	34	interface boundaries	40
object, pseudo	43	location transparency	35
object, righteous	43	mediation	39
object, target	34	pseudo object	43
omg		role	39
specifications	36	what constitutes	39
Operating Systems	10	ORB Modes	17
operating systems	10	orb, mediation by	39
orb		Organisation	xi
as an abstraction	40	OutputStream Class	115

P

Physical Memory Access	73	protocol	55
physical memory access	73	protocol, mutex implementation	69
PIDL	43	rtcorba type id	68
platform transparency	35	scheduling	65
POA		storage structure	69
active object map	45	Priority Banded Connections	61, 64
arguments	47	priority banded connections	61, 64
create	47	priority inversion	
functionality	44	defined	54
policies	46	priority inversion, bounded	67
rootpoa	45	Priority Machinery	65
POA Activation Methods with Priority	62	priority machinery	65
POA activation methods with priority	62	priority mapping, corba	68
Pools	64	priority mappings	58
predictability		Priority Model	62
distributed applications	54	Priority Models	59
real-time terms	51	priority models	59
rtos and	53	Priority Phenomena and Protocols	65
Predictability in Distributed Applications	54	priority, corba	68
predictability, distributed systems	54	processing context, client	42
Preparation	11	processing, request	49
Prerequisites	9	programming interface, orb pseudo object	43
priority		programming language transparency	35
associate	52	protocol configuration, client and server	61
associate inheritance protocol	69	proxies	38
corba to native	68	pseudo object	43
data	69	object adapters	44
inversion	65	orb	43
model	62	PIDL	43
native to corba	68	pseudo-idl	43
phenomena and protocols	65	pseudo-idl	43

Q

queue 63, 64 queue, assign to thread pool 63

R

Real-time 4
 real-time 3
 corba configuration 61
 corba current 59
 corba modules 57
 corba mutexes 60
 corba priority 58
 defined 51
 extension to java 71
 hard 51
 orb 58
 portable object adapters 62
 priority inheritance 60
 soft 51
 terminology 51
 triggers 52
 Real-time CORBA Configuration 61
 Real-time CORBA Current 59
 Real-time CORBA Modules 57
 Real-time CORBA Mutexes and Priority
 Inheritance 60
 Real-time CORBA Priority 58
 Real-time Extension to Java 71
 real-time mutex interface 70
 Real-time ORB 58
 Real-time Portable Object Adapters 62
 Real-time Specification 57
 real-time specification 57
 Real-time Systems 51
 real-time systems 51
 real-time systems, developing 52
 Real-time, What is 3
 reference, object 42
 character string 42
 definition 42
 representation transparency 35
 request
 assembling messages 39
 request processing 49
 requests
 delivering 41
 delivering to remote objects 39
 requests, delivering 41
 to remote objects 39
 reset (). 115, 116
 Resolving Servers 84
 resources, controlling 63
 righteous object 43
 role, client 38
 RTCORBA API Restrictions 18
 RTCORBA Current Interface 69
 RTOrb Run Scripts 82
 RTOS
 relevance in real-time 52
 rtos, real-time systems 52
 RTPOA 62, 64
 RTPOA Current 64
 RTPOA current 64
 Running OpenFusion CORBA Services 87

S

scheduling 58, 65
 scheduling, threads 71
 Scope of this Guide for RTOrb 5
 server 38, 42
 definition 42
 different terms 42
 role 38
 skeleton 38
 Server-side 91, 95, 102
 setargs() 107
 setBuffer () 115, 116
 setLittleEndian () 115

setWritePosition ().	116	strings, character	42
size ().	116	stub	37
skeleton	38	client	38
definition	38	definition	37
implementation instance	38	invocations.	37
implementation type	38	proxies	38
implementations.	38	surrogates.	38
server	38	stub, client	38
type	38	Sun Java Real-time System.	83
sockets	33	Sun Java Real-Time System Requirements . . .	81
Soft Real-Time Mode	17	surrogates	38
specification		Synchronization	73
corba	36	synchronization.	52, 73
mutex implementation	69	System and Environment Settings	81
stack.	55	system call execution times, bounded.	55
Standards	4	System Variables.	10
standards	4	system variables	10
strings	42		

T

target object.	34	threadpool, laned.	64
terminology	51	threadpools	60, 62
Testing the Installation	13	threads	62
The Real-time CORBA Mutex Interface	70	threadpools, and.	62
thread pool		Time- and Event-Triggered Systems	52
operation, basic mode	63	time- and event-triggered systems	52
Thread Pool Operation Basic Mode	63	transparencies	34
thread pool, queue assigned to	63, 64	transparency, location	34
thread pools		corba model	35
associate poa	64	orb	35
associations with rtpoa.	64	transparency, platform	35
Thread Scheduling.	58	transparency, programming language.	35
thread scheduling.	58	transparency, representation	35
Thread Scheduling and Dispatching	71	tuple.	53
thread scheduling and dispatching	71	type	
thread termination, asynchronous	73	skeleton	38
Threadpool Configuration.	27		

U

unbounded delays		language influence.	55
avoidance techniques.	55	priority inversion.	54
illustrated discussion of	65	Uninstalling.	13
interrupts	55	uninstalling RTOrb	13

