Spectra ORB
C and C++ Edition
Real-time User Guide

Version 2

Guide edition: 03 (02 May 2014)

Table of Contents

1 Preface

1.1 Conventions

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

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

Hypertext links are shown as *blue italic underlined*.

On-Line (PDF) versions of this document: Items shown as cross-references to other parts of the document, *e.g.* Section *1.2 Contacts* on page 6, behave as hypertext links: click the cross-reference to jump to that section of the document.

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

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

Extended code fragments are shown in shaded boxes:

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

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

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

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

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

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

1.2 Contacts

PrismTech can be contacted at the following contact points.

Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

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

Web: *http://www.prismtech.com*

Technical questions: *technical-support@prismtech.com*

Sales enquiries: *sales@prismtech.com*

2 Introduction

What is Real-time?

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

- ‘Immediate, as an event is occurring.’,
- ‘The actual time during which physical events take place.’
- ‘The processing and visibility of transactions and information as they occur, and not on a periodic or batch basis.’
- ‘...computer systems that update information at the same rate as they receive data...’

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.

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.

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 type 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.

3 How Spectra ORB Supports Real-time

The language and architectural components of Spectra ORB Real-Time address the practical issues of developing real-time applications for the real world. 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
- asynchronous event handling, transfer of control and thread termination
- interoperability and portability

4 Installation

This chapter describes how to install Spectra ORB SDR Real-time Plugin. Please follow the procedures carefully.

The installation files can be downloaded from the PrismTech Web site *http://www.prismtech.com/*

An installation file is typically simply a compressed directory archive. For UNIX platforms this can be a `tar.gz` file (a `tar` file compressed with GNU gzip) or a `tar.Z` file (a `tar` file compressed with the native compress). For Windows platforms this is a `zip` file.

4.1 Environment Variables

To build and run the example code provided with the distribution. Two environment variables are used, `EORBHOME` for the source installation directory and `EORBENV` for the target architecture.

See the html release notes for an up to date set of host and target build environments.

4.2 Conventions

The following conventions are used in this chapter:

- Commonly used directories are shown as:

`<EORBHOME>` - root directory of the ORB 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 semicolon (`;`) 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
```

4.3 System Variables

The `PATH` and optionally `LD_LIBRARY_PATH` environment variables must be set as described below.

The `PATH` must include:

- the directory where executables are located (`<EORBHOME>/bin/<EORBENV>`)

The `LD_LIBRARY_PATH` (for UNIX) or `PATH` (for Windows) must include:

- the directory where shared libraries are located (<EORBHOME>/lib/<EORBENV>)

4.4 Installation Procedure

4.4.1 General

All installed files are placed in the installation directory specified during installation - no files are stored in any of the UNIX or Windows system directories. The software can be completely removed from a system by simply deleting the installation directory.

4.4.2 Preparation

It is recommended that any existing installation be removed before installing the current version. To support multiple different versions simply install into a different directory or move the existing installation directory. Please note the following warning.

Uninstalling the ORB removes all the distribution's files, including the executables, license, configuration, and data files located in the various sub-directories. If these files are required, then they should be backed-up prior to uninstalling.

4.4.3 Extract the Distribution

On UNIX systems simply uncompress and 'de-tar' the installation file. For example, on Linux:

```
> tar xzf <file>.tar.gz
```

On Windows systems simply double click on the installation file to uncompress it, or explicitly uncompress it using a utility such as WinZIP.

4.4.4 Install the License File

A valid Spectra ORB license file must be placed into the <EORBHOME>/etc directory after has installation. Please note that the ORB will not run without a valid license file. PrismTech uses FlexLM for Spectra ORB product licensing.

License files are provided by PrismTech for services or products which have been purchased. Contact PrismTech for purchasing details.

Evaluation licenses are provided when a Spectra product is downloaded from the PrismTech Web site *http://www.prismtech.com/*.

4.4.5 Test the Installation

The installation can be tested by building and running the provided examples in <EORBHOME>/examples. Details of how do this are included in the html documentation.

4.4.6 Removing an Installation

The installation can be completely removed by simply deleting the installation directory after stopping any ORB based applications or services.

5 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.

5.1 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 mutates
- pre-emption to achieve greater determinism
- appropriate scheduling strategies

5.2 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.

5.3 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 behavior These characteristics are sufficient for soft real-time systems. Behavioral 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 behavior.

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.
- f*ault-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, optimized 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.

5.4 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

5.5 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 the *Bibliography*, on page *30*, 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++.

6 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 Spectra ORB Real-Time distribution's html pages.

6.1 Real-time Specification

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

Figure 1 shows the key Real-time CORBA entities. The features that these relate to are described below.

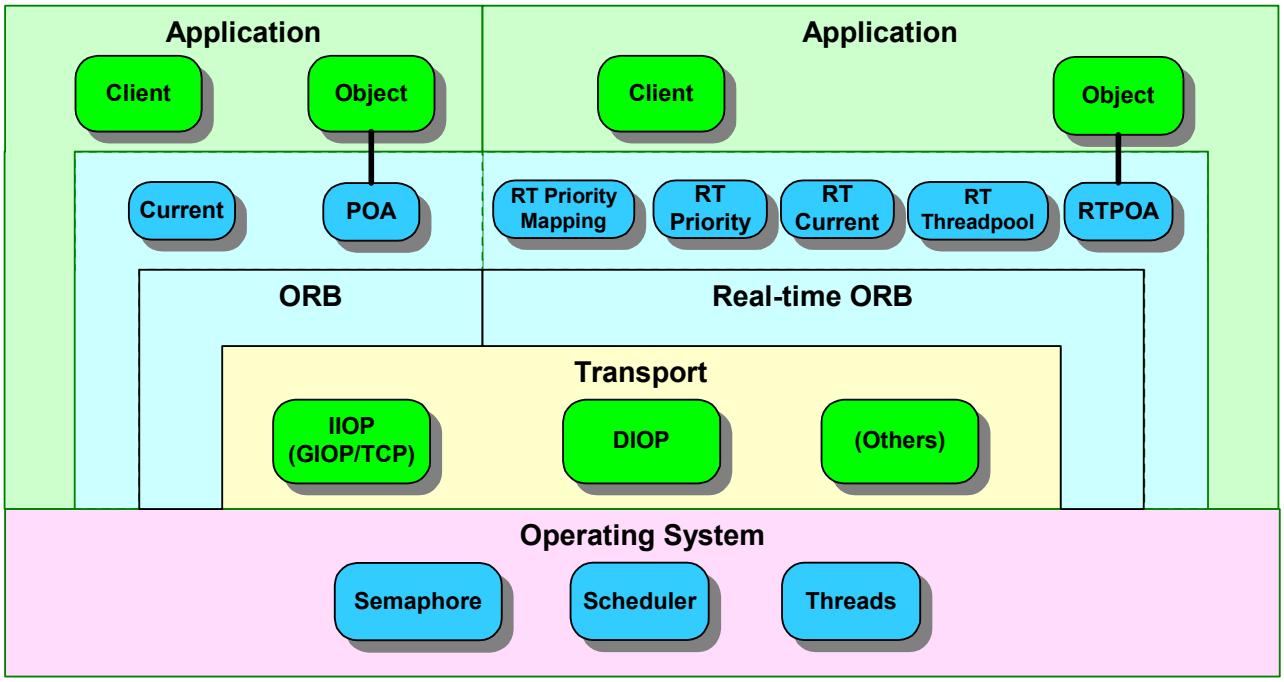

Figure 1: Real-time CORBA Components

6.1.1 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).

6.1.2 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.

6.1.3 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 Thread pool 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.

6.1.4 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.

6.1.5 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, 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.

6.1.6 Real-time CORBA Current

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

6.1.7 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.

6.1.8 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.

6.1.9 Thread Pools

Real-time CORBA uses the Thread pool abstraction to manage threads of execution on the server-side of the ORB. Thread pool characteristics can only be set when the thread pool is created. Thread pools 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 thread pool 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 thread pool 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.

```
module RTCORBA
{
   typedef unsigned long ThreadpoolId;

   struct ThreadpoolLane
```

```
   {
      Priority lane_priority;
      unsigned long static_threads;
      unsigned long dynamic_threads;
   };
   typedef sequence <ThreadpoolLane> ThreadpoolLanes;

   local interface RTORB
   {
      exception InvalidThreadpool {};

      ThreadpoolId create_threadpool
      (
         in unsigned long stacksize,
         in unsigned long static_threads,
         in unsigned long dynamic_threads,
         in Priority default_priority,
         in boolean allow_request_buffering,
         in unsigned long max_buffered_requests,
         in unsigned long max_request_buffer_size
      );

      ThreadpoolId create_threadpool_with_lanes
      (
         in unsigned long stacksize,
         in ThreadpoolLanes lanes,
         in boolean allow_borrowing,
         in boolean allow_request_buffering,
         in unsigned long max_buffered_requests,
         in unsigned long max_request_buffer_size
      );

      void destroy_threadpool (in ThreadpoolId threadpool)
         raises (InvalidThreadpool);
   };
};
```

6.1.10 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.

6.1.11 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.

6.1.12 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.

6.1.13 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.

6.1.14 Real-time CORBA Configuration

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

- server-side thread configuration (through Thread pools)
- 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.

6.1.15 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 thread pools, and priority banded connections.

6.1.15.1 Priority Model

Both the `RTCORBA::SERVER_DECLARED` and `RTCORBA::CLIENT_PROPAGATED` priority models are supported. Refer to the CORBA Priority Model example included in the real-time examples to see how to set the priority model policy for an RTPOA.

6.1.15.2 RTPOA

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

6.1.15.3 POA Activation Methods with Priority

```
module RTPortableServer
{
   local interface POA : PortableServer::POA
   {
      Object create_reference_with_priority
      (
         in CORBA::RepositoryId intf,
         in RTCORBA::Priority priority
```

```
      )
      raises (WrongPolicy);

      Object create_reference_with_id_and_priority
      (
         in PortableServer::ObjectId oid,
         in CORBA::RepositoryId intf,
         in RTCORBA::Priority priority
      )
      raises (WrongPolicy);

      PortableServer::ObjectId activate_object_with_priority
      (
         in PortableServer::Servant p_servant,
         in RTCORBA::Priority priority
      )
      raises (ServantAlreadyActive, WrongPolicy);

      void activate_object_with_id_and_priority
      (
         in PortableServer::ObjectId oid,
         in PortableServer::Servant p_servant,
         in RTCORBA::Priority priority
      )
      raises
         (ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);
   };
};
```

6.1.16 Threads and Thread Pools

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

6.1.16.1 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. Client applications use this mechanism to set the priority of the requesting thread.

6.1.16.2 Thread Pools

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 POAs. 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 POAs with thread pools where each POA associates with one or more thread pools (see Figure 2). Note that thread pools are independent of the POA lifecycle.

6.1.16.3 Thread Pool Operation Basic Mode

Application developers and end-users configure and control processor resources using thread pools (see Figure 2). Threads in the thread pool 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 life cycle.

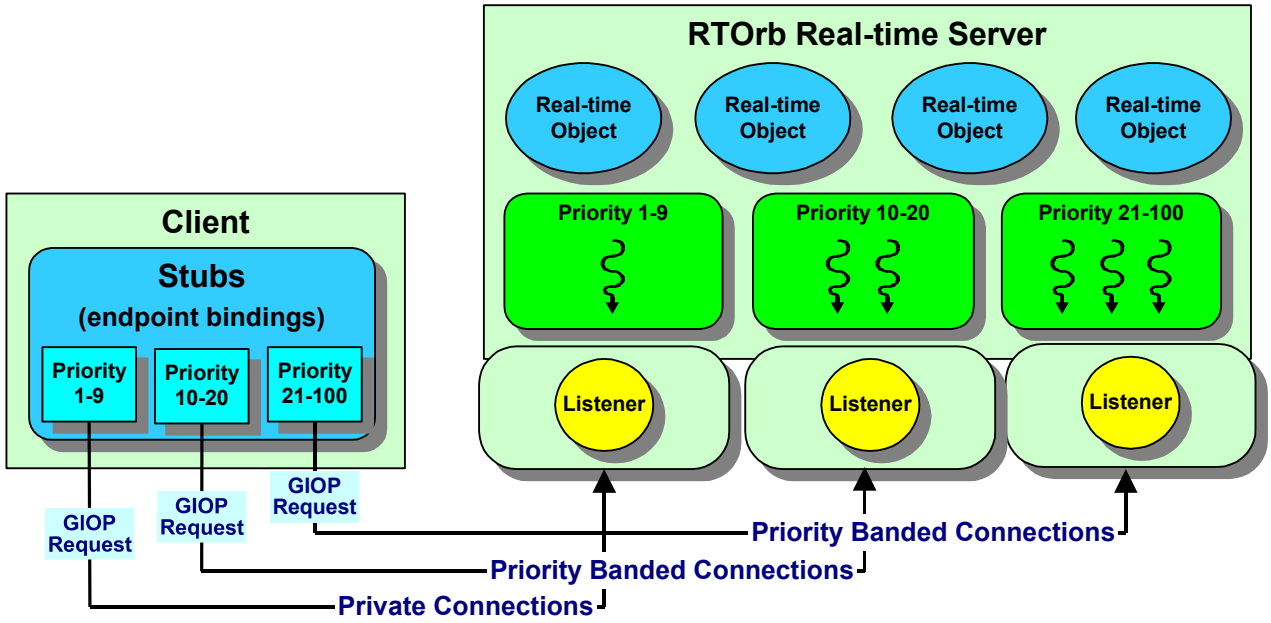

Figure 2: Thread Pools and Priority Bands

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.

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.

Thread pools can be configured for use with RTPOAs in one of two forms:

1. Non-laned Thread pool
2. Laned Thread pool

6.1.16.4 Laned Thread Pool

A thread pool 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.

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

6.1.17 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 requestor's 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 thread pools 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.

6.1.18 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 thread pool 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.

6.1.19 Priority Machinery

Priority is the medium used to achieve qualities of service in Real-time CORBA, hence the focus of Spectra ORB Real-Time application design. With the ORB 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 prioritized 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

6.1.19.1 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 *3*. Here, three tasks or threads are executing, T1, T2, T3. The tasks are illustrated in order of decreasing priority such that the priority of T1 is the greatest and that of T3 the least of the three. In addition, we assume that T1 and T3 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 *t0* task T3 starts to run. At time t1 task T3 locks and enters a critical section, continuing to execute until time *t2*. The portion of time for which task T3 is in a critical section is shown as shaded. At time *t2*, task T1 pre-empts task T3 because T1 has a higher priority. Task T1 now executes from time *t2* until time *t3*, at which point it attempts to gain access to the critical section, which has previously been locked by lower priority task T3. Task T1 is therefore forced to wait or block until such time as T3 releases the lock on the critical section shared between T1 and T3. Task T3 is allowed to run next. So at time *t3*, task T3 resumes execution and continues to work its way through the critical section.

Now at time *t4*, task T2 pre-empts task T3 and starts to run because task T2 has a higher priority than that of task T3.

Task T1 is now blocked by task T3 because of the shared resource, and task T3 is blocked by task T2. Therefore T2 is now indirectly blocking task T1 as well. Task T2 blocks task T3 until T2 completes at time *t5*. As a consequence T3 is forced to block for a significant amount of time (the length of the shared critical section *plus* the execution time of task T2).

For an actual system, when several medium-priority tasks exist with priorities greater than that of task T3 but less than that of task T1, it can lead to unbounded delay or blocking.

This effect is known as *priority inversion* and it occurs in the time interval *t3* to *t6*.

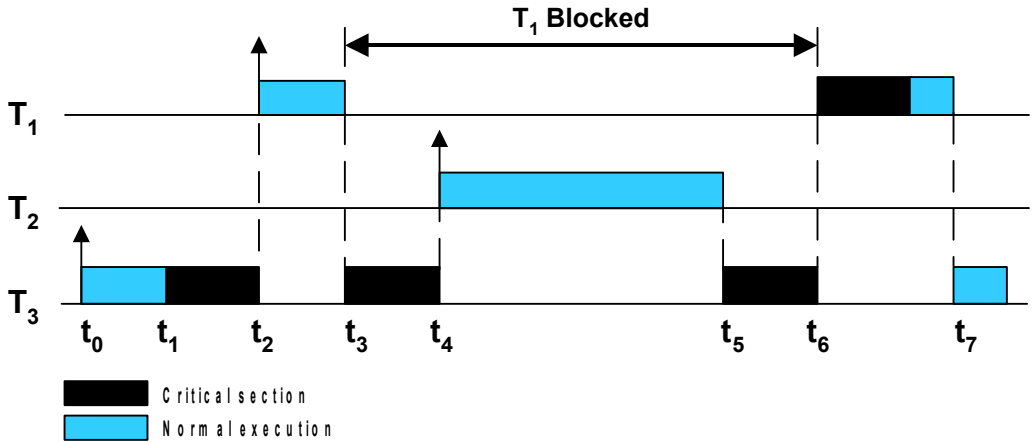

Figure 3: Priority Inversion

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 in the *Bibliography*, on page *30*, particularly Buttazo.

The Real-time Extension aids the RT CORBA developer by providing priority inheritance protocols in the ORB. Specifically, Spectra ORB Real-Time'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 4 and the following text explain how priority inheritance protocols bound any possible priority inversion. The same three tasks are illustrated as in Figure 3. Additionally, the relative priorities of the three tasks are depicted at the bottom of the figure as P1, P2, and P3.

Up to time *t3*, the behavior of tasks T1 and T3 are the same as in Figure 4. At time *t3*, T1 is forced to block on T3 due to T3 holding a lock on a critical section to which T1 needs access. At this point the mechanism of priority inheritance is employed. This mechanism causes T3 to inherit the priority P1 of task T1, which forces task T3 to execute immediately and run through the remaining part of its critical section. This forces T3 to execute from *t3* to *t5* at the T1 priority P1, which is the highest priority in this illustration. Note that task T2 cannot pre-empt task T3 as task T2 has lower priority than the temporarily-assigned priority (P1 of task T3, through priority inheritance).

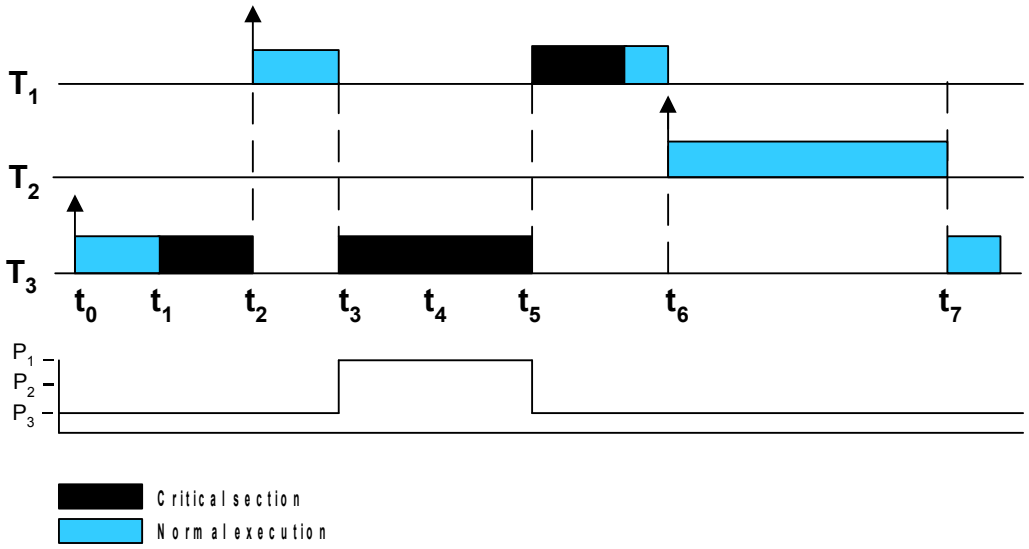

Figure 4: Priority Inheritance

As task T3 exits the critical section, its priority is returned to its original value P3 as shown in Figure 4. At time *t5*, task T1 can run because priority P1 is greater than priority P2 of task T2. Thus it no longer needs to block on task T3, which was holding a lock on the critical section. Task T1 now runs through the critical section and to completion at *t6*. At time t6, task T2 has the highest priority and executes as shown in Figure 4.

6.1.20 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.

6.1.20.1 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 from 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 RTCurrent get and set methods, and use of the default priority mapping.

6.1.20.2 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 as an IDL native, and allows you to forward and reverse map CORBA and native priorities as shown in Figure 5. Language-specific mappings exist for the priority mapping functions.

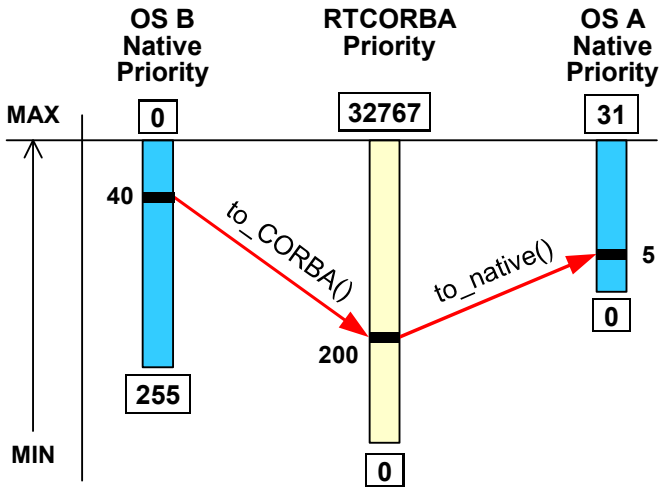

Figure 5: Priority Mapping

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.

6.1.20.3 CORBA Priority Transforms

Real-time CORBA supports the installation of user-defined Priority Transforms, to modify the CORBA Priority associated with an invocation during the processing of the invocation by a server. Use of these Priority Transforms allows application designers to implement Real-time CORBA systems using priority models different from either the Client Propagated or Server Declared priority models.

Priority Transforms exist in two forms. An 'inbound' transform can be applied prior to invoking a servant implementation. An 'outbound' transform can be applied when a CORBA invocation is made on an object. The transform interface is defined as an IDL native. Language-specific mappings exist for the priority transform functions.

6.2 CORBA Mutex

6.2.1 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.

6.2.2 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.

6.2.3 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);
   };
};
```

7 Implementation-Specific Features

The Spectra ORB RT CORBA ORB implementations include Spectra ORB-specific features. These features are present for one or more of the following reasons:

- The relevant CORBA specification does not specify how a feature must be implemented.
- To provide functionality beyond that currently specified by the OMG.
- To allow for flexible configuration of the ORB.

7.1 Supported Features

Not all real-time features are currently supported by the ORBs. The following table shows what features are supported:

Feature	C ORB	C++ ORB
Mutex	Yes	Yes
Priority Mapping	Yes	Yes
Priority Transform	Yes	Yes
Client Propagated Priority	Yes	Yes
Server Declared Priority	Yes	Yes
Threadpools	Yes	Yes
Threadpool Lanes	Yes	Yes
Threadpool Request Buffering	Yes	Yes
Threadpool Dynamic Threads	Yes	Yes
Threadpool Borrowing	No	No
Real-time Current	Yes	Yes
Real-time POA	Yes	Yes
Priority Banded Connections	Yes	Yes
Private Connection Policy	Yes	Yes
TCP Protocol Properties Policy	Yes	Yes
Client Protocol Policy	Yes	Yes
Server Protocol Policy	Yes	Yes
Thread Identity	Yes	Yes

7.2 Real-time Plugins

The C and C++ ORB feature a plugin system that explicitly controls what features are available in a given CORBA application. For real-time support the following plugins are available.

7.2.1 C ORB

Feature	Code required
Real-time ORB	`EORB_RTORB_plugin ()`
Real-time POA	`EORB_RTPOA_plugin ()`

7.2.2 C++ ORB

Feature	Code required
Real-time ORB	`EORB::Plugin::RTORB::add ()`
Real-time POA	`EORB::Plugin::RTPOA::add ()`
Priority Transform	`EORB::Plugin::PriorityTransform::add ()`

7.3 Priority Mapping Functions

The mechanism for using Priority Mapping functions is implementation specific. The following sections describe how to use Priority Mapping for the C and C++ ORBs.

7.3.1 C ORB

The C ORB provides two function pointers that can be set to user defined functions. The `RTCORBA_PriorityMapping_to_native` pointer should be set to a function that maps between a CORBA priority and a native priority.

The `RTCORBA_PriorityMapping_to_CORBA` pointer should be set to a function that maps between a native priority and a CORBA priority.

By default both function pointers are set to internal priority mapping implementations that do a simple linear mapping between the native and CORBA priorities. Neither of these functions should be set to `NULL`.

7.3.2 C++ ORB

In the C++ ORB the Priority Mapping function are defined by creating a subclass of `RT::CORBA::PriorityMapping`. This class has two operations. The `to_native` operation should map between a CORBA priority and a native priority. The `to_CORBA` operation should map between a native priority and a CORBA priority.

An instance of the subclass can then be used by passing a pointer to the instance to the static `set_mapping` operation on the `PriorityMapping` class.

Note that the default implementation of the C++ `PriorityMapping` class uses the C Priority Mapping mechanism hence it is possible to use the same pair of C functions to define a custom mapping for either the C or C++ ORBs.

7.4 Priority Transform Functions

The mechanism for using Priority Transform functions is implementation specific. The following sections describe how to use Priority Tranforms for the C and C++ ORBs.

7.4.1 C ORB

To set priority transforms in the C ORB simply set the priority transform function pointers defined in `RTCORBA/PriorityTransform.h`. These function pointers are `RTCORBA_PriorityTransform_inbound` for inbound priority transforms and `RTCORBA_PriorityTransform_outbound` for outbound priority transforms. By default both are `NULL` (not set).

7.4.2 C++ ORB

In the C++ ORB the Priority Mapping function are defined by creating a subclass of `RT::CORBA::PriorityTransform`. This class has two operations. The `inbound` operation should be used to modify a priority when a request is received by a server. The `outbound` operation should be used to modify a priority when a outgoing CORBA request is made.

An instance of the subclass can then be used by passing a pointer to the instance to the static `set_transform` operation on the `PriorityTransform` class.

7.5 Listener Thread Priority

The ORB transport uses a listener thread, for each supported protocol, to listen for new connection requests. There is no standard way to set this thread priority which may be an issue if this thread runs at a higher priority than an existing connection, where new connection requests may induce jitter.

7.5.1 C ORB

To set the default thread priority for listener threads an ORB initialization argument may be used:

```
-ORBListenerPriority <priority>
```

Where `<priority>` is a valid CORBA priority (`0` – `32767`).

7.5.2 C++ ORB

It is not currently possible to set the priority for listener threads on the C++ ORB.

7.6 Thread Identity

The ORB supports distributed thread identity via the `get_current_id` operation on the `RTScheduling::Current` interface. Internally this is implemented as a 32 bit unsigned long random value. As the `get_current_id` opeartion returns the identity encapsulated in an octet sequence, this can be used to actually set the value of a thread identity as the sequence contains a pointer to the actual id value.

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] *Patterns for Concurrent and Networked Objects*, Pattern Oriented Software Architecture - Volume 2, Schmidt D., et. al.,. J Wiley, 2000.

[3] *Predictable Scheduling Algorithms and Applications*, Hard Real-time Computing Systems, Buttazo G., Kluwer Academic Press, 1997.

[4] *Programming for the Real World*, Posix.4, Gallmeister B.O., O'Reilly and associates, 1995.

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

[6] *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kopetz, H., Kluwer Academic Press, Fourth Edition, 1997.

[7] *Synchronization in Real-time Systems: A Priority Inheritance Approach*, Rajkumar R., Kluwer Academic Press, 1991.

[8] *What is Predictability for Real-time Systems*, Stankovic J.A. and Ramamritham K., Journal of Real-time Systems, Issue 2, 1990.