Document for the Real-Time Message Passing Interface (MPI/RT-1.0) Draft Standard

Real-Time Message Passing Interface (MPI/RT) Forum
http://www.mpirt.org

June 7, 1999

Contents

Preface

In 1995, several researchers and practitioners became interested in advancing real-time extensions to the then existing Message Passing Interface (MPI) de facto standard and began meeting informally, and later, as a sanctioned subcommittee of the MPI-2 de facto standards body, which met regularly in Chicago. People involved with high-performance computing, distributed computing, message passing systems, and real-time systems were all represented. Included here were researchers and practitioners from industry, academia, and defense laboratories. The MPI-2 Forum condoned this effort and allowed it to blossom as a "Journal of Development" activity, with a clear view that it would not be formally part of MPI-2, but nonetheless a worthwhile working activity. This status was productive and helpful to the work because of the valuable proximity to many interested in messaging, without the compelling deadline faced by MPI-2.

From a technical perspective, a lot of issues, approaches, requirements, and techniques evolved, and significant new ideas previously thought about were introduced into the discussion. These issues included provision of key concepts not available or readily addressable within the confines of MPI-1 and MPI-2: channels, real-time models, predictability, greater support for thread interactions, early-binding strategies, and admission tests. The requirements posed by the subcommittee emerged as follows: achieve highest performance messaging and add the additional constraints of predictability and quality-of-service, with the additional capability to support relevant memory management to enhance the elimination of data copies and support for "cut-through" of data.

These requirements drove us over time, sometimes systematically and sometimes ad hoc, to reexamine much of what was decided in earlier messaging systems and ultimately to evolve away from explicit upward or downward compatibility with MPI. The outcome of this effort is a "lower middleware" standard called MPI/RT, which strives to offer extremely low cost of portability as compared to any native software architecture for messaging, while providing useful real-time notions of performance, predictability, and quality-of-service. Other major decisions included elimination of Fortran77 language bindings in favor of C++ language bindings and the thorough and continued use of object-oriented API's and design methodologies to motivate and support the process. Positioning MPI as conceptually "higher middleware" in the form of a layer on top of MPI/RT establishes a conceptual relationship between this work and the previous standard. In fact, we expect that MPI implementations will actually be layered over MPI/RT on systems where users require both notations, and this layerability is mentioned in appropriate parts of the standard.

With encouragement and support from DARPA, and from the strong commitment of many of the subcommittee participants, including people from the mainstream of MPI forum participants, significant progress was made over the first eighteen months. The group continued to meet and burgeoned into a full-scale de facto effort of its own after the conclusion of the MPI-2 standards effort. Early versions of this effort appear in the "Journal of Development" of the MPI-2 standard, but the results presented in that snapshot are quite different from what we have ultimately accomplished.

This three-year effort has led to quite a satisfactory messaging middleware specification and standard that we expect to see deployed by industry in real-time computing multicomputers and networks of workstations. The group is committed to extend the specification in a limited fashion in 1999 to support channel I/O, dynamic processes, and a few other features intentionally delayed at present. After that, sincere efforts to introduce MPI/RT to a formal standard body will be undertaken by us and others, in order to help assure its long-term acceptance.

In order to facilitate accessibility to this document, we have explicitly moved all references to the MPI standards within the document into footnotes, in order to provide additional information on relatedness to MPI-informed people, while keeping the main flow and standardized content of the document fully free of dependencies on either MPI-1 or MPI-2 standard documents. No knowledge of MPI-1 or MPI-2 is needed to proceed with MPI/RT. In this way, we can mention issues where credit is due or different decisions have been made, while avoiding dependence on the other documents moving forward, and without requiring big background study from new arrivals to MPI/RT.

As part of this work, we have generated our own "Journal of Development" that has been divided into two classes of results: first, there is material to be revisited in the MPI/RT-1.1 version, beginning in late 1998. Second, there is "best engineering practice," results and issues worked out during the past years that we want to preserve. A separate, but not formally accepted, best engineering practice document is consequently a related outcome of this group's work. Such issues as interoperability, relationships to other standards, and a standardized subset of MPI/RT-1.1 figure in that other document. This other body of results remains separate from, yet valuable to, the community we are seeking to support.

The complete specification for MPI/RT is presented in this document. Chapter 1 provides a high-level overview of the standard for first-time readers. Chapter 2 presents the object-oriented design of MPI/RT and describes the fundamental ancestor objects in MPI/RT's class hierarchy. Chapters 3 through 5 describe the basic datatypes, the event handling mechanism and buffer management. Chapters 6 through 9 describe the channel-based data transfer operations. Chapters 10 through 12 discuss real-time Quality of Service issues and admittance testing. Finally, Chapters 13 through 15 address the environmental issues for MPI/RT programs such as initialization, termination, synchronized clocks and performance monitoring. Material of special interest to users, such as a detailed description of how to use a particular feature of MPI/RT, is highlighted by *Advice to users.* Material of particular interest to implementers of MPI/RT is highlighted by *Advice to implementers.*

Arkady Kanevsky, Bedford, MA
Anthony Skjellum, Starkville, MS
July, 1998

Acknowledgements

Technical development was performed by the MPI/RT Forum members. During the forum meetings, several people served in positions of responsibility, and are listed below.

- Anthony Skjellum (MSU), Arkady Kanevsky (MITRE) — Chairs
- Yoginder Dandass (MSU) — Minutes, Document and Web Page Maintainer
- Anthony Skjellum, Arkady Kanevsky, Yoginder Dandass, Shane Hebert (MSU), Greg Henley (MSU), Zhenqian Cui (MSU and MSTI), Anna Rounbehler (Raytheon) — Editors
- Greg Henley, Shane Hebert — Initial Implementation

Other major contributors to the standard efforts, who were regularly involved in contributing time, content, and elbow grease, are as follows (alphabetically):

- Darwin Ammala, MPI Software Technology, Inc.
- Ronald Brightwell, Sandia National Laboratories
- Dennis Cottel, SPAWAR Systems Center, US Navy
- Robert Cunningham, MIT Lincoln Laboratory
- Nathan Doss, Sanders / Lockheed Martin GES
- Michael Grieco, NSA
- James Lebak, MIT Lincoln Laboratory
- Leonard Monk, MITRE and CSP, Inc.
- Steve Paavola, SKY Computers
- Thomas Robey, Khoral Research
- Jerrell Watts, Caltech and Syracuse University

Following is a list of active participants in the MPI/RT Forum not already mentioned above:

Tarek Abdelzaher	Manoj Apte	Daya Atapattu
Robert Babb	Peter Brennan	Keith Bromley
Chuck Browne	Mike Butler	Billy Chin
George Crawford III	Tom Daggett	Marc Fidler
Joe Fogler	Bob Ford	Richard Games
Robert George	Joe Germann	Robert Ginn
Clair Guthrie	Chris Hayden	Jon Hiller
Eric Holbrook	Paul Husted	Jim Ionata
Randall Janka	Gabriel Jones	Randall Judd
Tony Lee	Lloyd Lewins	Mark Lin
Thomas McMahon	Harish Nag	John Ramsdell
Stephen Rhodes	Brian Schott	Donald Shakley
Tim Singleton	Alan Skillicorn	Rob Steele
Richard Steinberger	Clay Taylor, Jr.	Gerard Vichniac
John Williams		

The following institutions were represented in one or more meetings of the Forum:

Advanced System Architectures	CSP, Inc.[1]
Denver University[1]	GTRI
Hughes	Integrated Sensors
Intel SSD	Khoral Research
Lockheed Martin/GES[1]	Mercury Computer Systems
MHPCC/AHPCC[1]	Mississippi State University[1]
MIT/LL[1]	MITRE[1]
MPI Software Technology, Inc.[1]	NAWC
NSA[1]	NUWC
Raytheon[1]	Sanders[1]
Sandia National Laboratories	Scientific Computing Associates
SKY Computers[1]	SPAWAR Systems Center[1]
Science and Technology Associates, Inc.	Syracuse University
US Navy/TASP[1]	USC/ISI

The MPI/RT Forum thanks the following institutions for hosting the meetings: The MITRE Corporation, MSU, SPAWAR System Center, Sandia National Laboratories, and Science and Technology Associates, Inc. (STA).

Mississippi State University provided computing resources for maintaining the drafts of the document, the Web pages, mailing list, and played an essential role in distributing the document.

The Defense Advanced Research Projects Agency (DARPA) provided major funding to Mississippi State University through the United States Air Force Research Laboratory, Rome, NY, under contracts F30602-95-1-0036 and F30602-96-1-0329 for conducting research towards realization of the MPI/RT-1.0 standard, for organizing and conducting the standards meetings, for preparing and disseminating the document, and for work on the Inertia and Momentum prototypes.

This work was supported in part by the U.S. Air Force Electronic Systems Center and performed under MITRE Mission Oriented Investigation and Experimentation Project 03987450 of contract F19628-94-C-0001, managed by The Air Force Research Laboratory/IFTC.

[1]Official voting institution at the conclusion of the MPI/RT-1.0 standard.

Acknowledgement is due to Dr. Jóse Muñoz of the DARPA Information Technologies Office (ITO), for fully supporting MPI/RT as a next-generation parallel programming concept, research effort, standard, and finally, as an emerging COTS technology.

Additional support was received at Mississippi State University from the Tactical Advanced Signal Processor (TASP) project of the U. S. Navy, for additional work on MPI/RT and Inertia. Messrs. Tim Singleton, Clair Guthrie, and Jim Broughton of the U. S. Navy are acknowledged for their continuing support for this standard. Dave Toms, formerly with U. S. Navy, is also acknowledged for his support involving TASP and MPI/RT. ⊥ (final)

Glossary

The following is a glossary of terms used in this document. This includes both terminology and acronyms relevant to MPI/RT. Wherever appropriate, the definitions provided by IEEE [8] and Software Engineering Institute [3] are used. ⊤ (final)

Admission Test The test performed to determine if the requested quality-of-service for all the objects requested by the user can be satisfied. This test can be performed on-line or off-line. The test can produce the schedule for the object operations. In general, the admission test is separate from the resource allocation and the transitioning of objects into a committed state. This is an integral part of the MPIRT_COMMIT operation. ⊤ (final2)

Application Phases One of four phases an application can be in with respect to MPI/RT. The four phases are: startup, setup, real-time, and cleanup. ⊥ (final2)

Application Programmer Interface (API) The specification of a set of functions which defines the interface to a piece of software or a software library.

Application Process A unit of work that is an application program (user code).

Buffer A portion of memory designated to contain data as it transits from one point to another within the system. It is useful in cases where uniformity in processing speed or transfer speed is lacking.

Buffer Iterator A mechanism to achieve ordering of buffers associated with an endpoint of a channel. This ordering is user specified and defines the order in which buffers are consumed and returned by the channel endpoint.

Buffer Object An opaque representation of a buffer containing buffer parameters, such as size and dataspec of the buffer's contents, the address of the buffer, and a reference for the object itself.

Cached Attribute An attribute of an object which is retained for more expedient use by MPI/RT objects or in duplication of objects.

Channel A persistent, logical conduit by which data travels from one process to another. Constituent parts are sending and receiving endpoints, buffer iterators, quality-of-service, name, rank, and group.

Channel Endpoint A local part of a channel where data either enters or exits. It consists of a pair of buffer iterators for all channels and one more pair for collective channels for which the local process is involved in both sending and receiving data. ⊤ (final2)

Cleanup Phase The phase of an application where an application is terminating MPI/RT implementation.

Collective Operation An operation which must be performed by all members of a group.

Collective Channel A persistent logical conduit by which data travels between all members of the group. A collective channel can be logically viewed as a conglomeration of point-to-point channels used by a collective operation. They may be comprised of point-to-point channels or a specific network. They are composed of input buffer iterators, output buffer iterators, group, quality-of-service, and collective name.

Commit Operation The commit operation reserves resources for MPI/RT objects and the implementation according to the admission test to satisfy user quality-of-service requests and switching control according to a schedule to satisfy these quality-of-service requests.

Committable Object An object which undergoes a two phase initialization. These phases are CREATE and COMMIT. Through this process MPI/RT achieves a deferred early binding.

Condition A transition of an object from one state to another. Users and implementation can request a notification when a condition occurs via event receptors. If one or more receptors are specified for an implicit event defined for this condition, they will raise the event trigger associated with this condition. If an event receptor is specified for a condition based on an event name, a trigger is created implicitly by an implementation that will be raised when the condition occurs. Conditions are specified in the state diagrams in this document.

Container An MPI/RT class of objects which may hold a collection of MPI/RT objects.

Create Operation The first phase in an object's life cycle where user requirements for an object and interactions between objects are defined. The result of the create operation is an object descriptor.

Data Descriptor The data descriptor consists of two parameters: *d*ataspec that describes the format of data and *b*ufsize that represents the size of data in the number of the dataspec elements. The data descriptor is part of the buffer and the buffer iterator objects specification.

Data Transfer Operations Operations defined on channels that allow users to transfer one or more messages. These operations support two-sided, one-sided or zero-sided communication paradigms.

Dataspec MPI/RT data types that can be transferred between processes via channels. Dataspecs are used in the definition of the buffers. The dataspecs sometimes are referred to as "transmittable types".

Deadline A point in time by which an operation or an activity must be completed for timing behavior of the system to be correct. A deadline is the part of the quality-of-service that specifies when an operation must be completed. In comparison, a *timeout* specifies a point in time when a user should be notified if a certain action or operation has not been completed, and that action or operation should be stopped by an implementation.

Deadlock The condition of a system in which every response in a set of responses is using a resource and is also waiting for another resource that must be first relinquished by another response in the set.

Deferred Early Binding A mechanism through which system and user resources may be allocated during execution versus a priori. It is accomplished in MPI/RT via the two-phased commit process. An object descriptor is established upon creation, and the object-body is instantiated during the commit operation.

Dup Operation An operation (duplication) to create a copy of an existing MPI/RT object.

Environment An environment consists of all the activities and devices outside user control. These include computational platform, devices, and outside environment.

Event An occurrence of a noteworthy happening (user, system, MPI/RT) that is of potential interest to a user.

Event Kind All the events that share the same event name and the scope.

Event Delivery Abstraction A mechanism for delivery of events of interest to an implementation and/or a user through MPI/RT. It contains two independent parts: triggers and receptors that are matched by an implementation based on the event name over the scope of the event. This abstraction supports a discrete notification to MPI/RT that a certain condition (user, system, MPI/RT) occurred via raising event triggers. MPI/RT in turn delivers the event notifications from triggers to receptors with the requested quality-of-service parameters. $\top$ (final2)

Event Scope A set of processes that specify the scope of the delivery of an event from a trigger to a receptor. The event is delivered from the trigger to the receptor that share the event kind name if and only if both the trigger and the receptor processes belong to the event scope. $\perp$ (final2)

Free Operation The operation which returns allocated system resources associated with an object and releases the object body and descriptor.

Functor An object used to encapsulate information from the user and the implementation to perform some function or operation. A handler object is an example of a functor.

Get Operation An operation to retrieve parameters or attributes of an object.

Group An ordered collection of processes represented as MPIRT_TASK_ADDRESSes in the vector base container. In comparison to its sibling MPIRT_CVECTOR, MPIRT_GROUP resizes itself automatically to remove empty (MPIRT_OBJECT_NULL) elements.

Handler An invokable user or predefined function launched by an implementation based on event arrival.

In Argument An argument that is passed from an application to an implementation.

In/Out Argument An argument that is passed from an application to an implementation and back.

Inter-trigger Event Time A minimum time between generation of two successive events of a trigger.

Language Binding A mapping of LIS into some specific programming language. This document provides language bindings for C and C++.

Layered Library An Applications Programming Interface which supports a function or environment. Examples are MPI-1, MPI/RT, and VSIP. By their design, they can coexist within a user application or system.

LIS Language Independent Specification.

Memory Descriptor A description of some memory for communication purposes. A buffer is a simple, contiguous version of it.

Middleware The portion of a system which interfaces the user's code to the resources of the system.

Object A logical, unitary structure of attributes, parameters and data which serves a specific purpose within MPI/RT.

Object-body The parameters and data portion of the object. It is a physically distinct portion of the logical object entity.

Object Decoration See "cached attribute."

Object-Descriptor A physically distinct part of the logical object in the system space which contains the parameters for the object-body with which it is associated.

Opaque If an object is opaque, none of its parameters can be manipulated by the user. Users can manipulate the data portion of an object indirectly by using operations defined in this standard.

Out Argument An argument that is passed from an implementation to an application.

Persistent Object Having the property of existing longer than a single operation. All the objects in MPI/RT are persistent.

Priority A value associated with an action that is used by an implementation for allocating resources to resolve contention for shared resources. It specifies a measure of comparative urgency in scheduling an operation. The most urgent operations are scheduled at any given point in an application.

Probe An instrumentation object used by the programmer to retrieve metric values from the implementation or from a user metric.

Progress Rule The rule that governs when data transmission should occur. Stated simply, data transmission between channels should occur even if no MPI/RT function is called to return control to the implementation. Scheduling, threads, and interrupts can be used to achieve this effect[1].

Quality-of-Service Quality-of-service (QoS) consists of user-requested dependencies, constraints, and requirements that are specified by the combination of a *requirement* and a *constraint* for an object. An admission test determines if an object's quality-of-service can be satisfied by the system.

[1] This is similar to the Progress Rule of MPI.

Quality-of-Service Constraint (QoS-Constraint) A part of the quality-of-service specification which is the user's requested constraints on a start and/or finish for an object operation. The constraint is specified by event receptors or time instances in the form of timeouts that guard an operation time window.

Quality-of-Service Requirement (QoS-Requirement) A part of the quality-of-service specification which is the user's requested requirements on a start and/or finish for an object operation. The requirement is specified by the starting time, deadline or priority.

Raise Operation This operation allows users to create and notify an implementation of an occurrence of an event for an event trigger. ⊤ (final2)

Real-Time Mode A real-time mode is a state the application is in when it is in real-time phase. The real-time mode is defined by the set of objects provided to the implementation in commit operation. In real-time mode all the objects provided to the implementation at commit time are guaranteed to have their quality of service specification satisfied by the implementation.

Real-Time Phase The real-time phase is one of the four phases of an application where application can use system resources provided by an implementation for user-defined MPI/RT objects and the implementation supports primary operations on objects like channel message transfer, event delivery, handler execution and passing buffers between the application and the implementation. Within the real-time phase an application can transition from one real-time mode to another using the commit operation. The real-time phase is preceded and followed by the setup phase. ⊥ (final2)

Receptor A receptor is a part of the Event Delivery Abstraction. This is a notification from the system to the user or MPI/RT object that something of interest to the user has occurred. A receptor is an instantaneous mechanism that schedules an execution of one or more handlers that are defined for the receptor.

Retrieve Operation An operation to obtain a reference for an object or a parameter of an object that can be a reference to another MPI/RT object. ⊤ (final2)

Setup Phase The phase of the application where the application is creating and is destroying MPI/RT objects. ⊥ (final2)

Set Operation The operation to define or edit an attribute of an object.

Set Container A resizeable container which holds references to zero or more MPI/RT objects in no certain order and which contains no duplicates. Also called S-Container. ⊤ (final2)

Startup Phase The phase of an application where an application is initializing MPI/RT implementation. ⊥ (final2)

Synchronization A process of coordination between application processes.

Synchronization Point An instance of synchronization between application processes. Synchronization can be achieved in MPI/RT explicitly via a barrier collective channel or implicitly using an event-driven model.

Synchronized Clock A system clock which has been synchronized with an external time source and between all processes. This clock serves as the MPI/RT global master clock from which the time-related functions read the time.

Task Address is an MPI/RT object that represents a computation unit (process) and which forms the elements of the MPIRT_GROUP. These objects are automatically created by MPIRT_INIT and can not be created by the user. This object supports free, duplication, equality testing operations, and attribute functionality.

Timeout Specifies a point in time when a user should be notified if a certain action or operation has not been completed. It is a completion constraint, with respect to the start of an operation, which poses no requirement for MPI/RT.

Transmittable Types See "Dataspec."

Trigger A trigger is a part of the Event Delivery Abstraction. An explicit trigger is a notification from a user to MPI/RT that some event of interest to some user process has occurred (has been raised). An implicit trigger is a notification from the system (including MPI/RT) to MPI/RT that some event of interest to some user process has occurred.

Types Data types are defined by MPI/RT to represent various memory constructs that are either not standard for all languages or not provided by some languages. For portability reasons MPI/RT defines some common operation for these data types.

Vector Container A resizeable container which holds references to zero or more MPI/RT objects or integers, has an index order $0, \ldots, N-1$, for N entries, and has random access to any entry in $O(1)$ time. In comparison to its sibling MPIRT_GROUP, MPIRT_CVECTOR does not resize itself automatically to remove empty elements, and can store empty elements (MPIRT_OBJECT_NULL).

Vector Base Container The vector base container is a virtual class is a resizeable container which holds references to zero or more MPI/RT objects or integers, has an index order $0, \ldots, N-1$, for N entries, and has random access to any entry in $O(1)$ time. Also called V-Container. The vector base container object has two children classes: MPIRT_CVECTOR and MPIRT_GROUP.

$\perp$ (final)

Chapter 1

Introduction

1.1 General Introduction

The state-of-the-art of real-time programming currently leads most application designers and programmers to become overly concerned with their immediate target platform in order to make even reasonable use of its communications features. This occurs substantially because quality-of-service is most often obtained empirically, indirectly, and/or with reference to specific hardware mechanisms and entrenched middleware policies that encapsulate these mechanisms. This approach to programming parallel real-time applications is entirely nonportable and have led to an unfortunate, expensive set of legacy codes that must often be redeveloped or replaced when system hardware or software upgrades are performed.

The tight coupling between application and platform makes it difficult to port existing applications to upgraded or new platforms and to alternative configurations within the same platform, exacerbating cost of ownership for such systems. In the common case, both syntactic/semantic correctness of program formulation and timing correctness must be rethought in the process of porting.

MPI/RT works to address real-time concerns by describing how various system resources, such as the network, memory management, and communication scheduling, can be marshaled (consistently with a runtime environment that allows MPI/RT to mold policy of the composed system) to design parallel applications around MPI/RT's service paradigm. MPI/RT provides an application-programmer-interface (API) whose design includes the ability to "plan in advance," "reserve resources," "admit sets of channels with timing guarantees," and to exploit temporal locality associated with programs that, while they possess several modes of operation, exhibit regular (e.g., repetitive or periodic) properties in a given mode of operation. For such applications, advance planning with concomitant resource reservation can have substantially positive impacts on both predictability and absolute performance. MPI/RT addresses the portability concern of syntax and semantics of services completely and formalizes and organizes, for the application developer, that which is necessary to re-achieve timing correctness in the face of system upgrades or replacements.

MPI/RT adheres to the underlying assumptions of its predecessors[1] of reliable message transfers and maintaining of the order of message delivery between the same pairs of sources and destinations. But in the case of ordered message delivery this order is subject to the channel quality of service specification over which the messages are delivered.

The primary design philosophy governing MPI/RT development is that only its implementors need to become expert users of platform-dependent communication features. Application devel-

[1] MPI-1 and MPI-2

opers need only supply the communication structure and required quality-of-service (QoS); the MPI/RT middleware orchestrates disparate subsystems and system APIs and provides translation of relatively high-level quality-of-service specifications into mechanisms suited to the underlying platform. MPI/RT will either achieve admission of the required services or reject the proposed real-time mode. In the case of rejection, the MPI/RT implementations will provide specific instrumentation and other approaches to help users achieve subsequent successful admission, possibly including off-line simulation and optimization.

1.1.1 Parallel Models

MPI/RT supports three parallel programming models, all of which assume reliable transfer: data-parallel (usually single-program, multiple data, or SPMD) involving groups of communicating processes, task parallel involving sets of data parallel groups (some of which may degenerate to be singletons), and coarse-grained dataflow comprising cascades of data-parallel groups. The fundamental abstraction[2] – the static process group, proves sufficient to describe sets of processes that overlay on a hypothetical, completely connected graph of processes. From this "template," MPI/RT programs describe point-to-point and collective channels that support actual transfers and embeddings in the graph. The channels not only play the role of a persistent virtual connection between a pair or a group of processes but also play the role of communicators that define a communication domain. QoS-based communication derives from channels that connect processes within groups either in pairs or in collective roles.

MPI/RT describes, in detail, the behavior of individual channels and collective channels and requires users to make use of patterns of communication in order to support data motion. MPI/RT is also more geared toward exploiting application temporal locality with concomitant advantages in performance and predictability for "regular" application models. MPI/RT allows the user to orchestrate operations toward guaranteeing specific system behavior and toward avoiding unneeded all-to-all virtual connections between mutually disinterested processes. For the latter reason, MPI/RT stands out as a scalable message passing standard for extremely large parallel systems. However, this scalability comes at the price of early binding organization and detailed program design.

MPI/RT provides a single, static process world, which may be decomposed into sets of overlapping and non-overlapping groups[3].

MPI/RT does not provide programming models such as virtual shared memory, nor cache-coherent, non-uniform memory access (CC-NUMA), nor other features offered by systems to simplify the appearance of the distributed system or to give the appearance of "single system image." MPI/RT has anchored itself in explicit parallel programming in which synchronization is implicit and carried with the transfer of data, but in which data transfer is always explicit. MPI/RT supports two-sided, one-sided and zero-sided communication, all through channels. The modes of operation over channels provide clear state transitions on how transfer is accomplished, without specific connection to the mechanisms that are utilized in order to achieve transfer effectively. Both the one-sided and zero-sided modes effectively support both push- and pull-oriented communication. The zero-sided mode involves no specific recurring interaction with the user, other than the filling and emptying of message buffers at appropriate times. This is representative of a powerful type of planned transfer possible within MPI/RT.

[2] Borrowed from MPI.

[3] MPI, the predecessor of MPI/RT, relies heavily on communicators (static process groups plus context-mux/demux information for messages) in order to support programming with composition of libraries and lexically-scoped messaging. MPI/RT de-emphasizes the communicator because MPI/RT relies on channels, which themselves provide context of communication.

MPI/RT does not standardize the way in which processes come into existence or are killed, and thus skirts the difficult and thorny issues of *fork()* and *exec()* dealt with by POSIX [9]. MPI/RT specifically supports the notion of a set of computers in which MPI/RT has complete control of resources, and whose operating systems schedule both the user application program(s) and the MPI/RT communication middleware according to MPI/RT requirements in order to achieve the quality-of-service desired. Highly dynamic or unpredictable real-time concerns are not presently addressed by MPI/RT. ⊤ (final)

1.1.2 Sidedness of Communication

MPI/RT provides support for three types of sidedness of an application message passing.

The first one, and the most commonly used one, is *two-sided* communication. It is characterized by the application issuing data transfer operations for two sides of the data exchange (all sides for collective operations). This paradigm is commonly called *send-receive*. This can be achieved in MPI/RT by calling MPIRT_CHANNEL_START operation on all the channel endpoints.

The second – *one-sided* communication type allows only one side of the application data exchange to issue data transfer operations. The most commonly used models of one-sided operations are *push* and *pull*. MPI/RT supports both of these models. Any one endpoint of a channel can issue MPIRT_CHANNEL_START operation while other channel endpoints can issue MPIRT_CHANNEL_ACTIVATE to specify that they will not issue any data transfer operations.

The last sidedness type is *zero-sided* communication. It is characterized by the absence of any data transfer operation by all sides of the data exchange. It is the responsibility of a communication service to move the data from/to prespecified application memories, at prespecified times, or in response to prespecified events. With the pre-established early binding channels MPI/RT exploits "no-sided" communication where the middleware (MPI/RT implementation) does the data transfer operations on behalf of the application at the predetermined times or in response to the predetermine events that are part of the channel QoS. A user notifies MPI/RT that the application will not issue any channel data transfer operations for all the channel endpoints via MPIRT_CHANNEL_ACTIVATE and guarantees that the message buffers involved in the channel transfers will be available for MPI/RT prior to the transfers in accordance to the channel QoS. ⊥ (final)

1.1.3 Real-Time Models and Quality-of-Service

At the outset, the MPI/RT working group sought to support "common practice" from the real-time parallel programming world. This was roughly determined to include time-driven, event-driven, and priority-driven real-time paradigms. Over the period of standardization effort, certain articulation of these real-time paradigms has occurred:

- The time-driven, real-time paradigm specifies timing intervals during which message transfers over channels should take place. This specification is used for the QoS of the channels only. This paradigm specification also includes periodicity of message transfers. Time instances for a time interval are specified either as the absolute time instances in the synchronized clock values or as the relative time instances. For the relative time instances, a starting time (beginning of the timing interval) is relative with respect to the beginning of a period, and a deadline (end of the timing interval) is relative with respect to the starting time of the timing interval.
- The combined event-driven-with-priority real-time paradigm specifies priority as a real-time requirement and events for the start and the completion (if an activity has not finished yet) for a channel message transfer, handler execution, or event delivery from a trigger to a

receptor. Individual messages do not possess priority per se, but "inherit" a channel priority that is specified as the channel QoS. Applications that seek multi-priority levels can construct multiple "parallel" channels with different priorities to achieve such formulations.

- The combined event-driven, time-driven-with-priority real-time paradigm specifies either a priority as the real-time requirement and a timeout as the completion event or a deadline as both the real-time requirement and the completion event for a channel message transfer or event delivery from a trigger to a receptor. Both forms can specify a starting event. The deadline, when it is used, is specified as the time instance relative to the starting event.

1.1.4 Ontogeny of an MPI/RT Application

⊤ (final2)

This section describes the phases of an MPI/RT application and emphasizes MPI/RT's deferred early binding protocol, in which the application is required to set up the communication architecture and its parameters, but the actual resource allocation is performed when the application enters its real-time phase.

The first phase of MPI/RT application is initialization. During this phase an application cannot use any MPI/RT functions except MPIRT_INIT to initialize MPI/RT and inquiry functions MPIRT_IS_INITIALIZED and MPIRT_IS_FINALIZED to determine the state of MPI/RT. The various processes that will participate in real-time communications are started in this phase; standard is silent as to how they are started. Implementations may choose to provide this functionality as a part of an *mpirtrun* executable[4].

The second phase is the real-time setup phase, which occurs once the MPIRT_INIT function returns successfully. In this phase, the application creates all MPI/RT objects with their requested QoS. System resources for these objects are not allocated during this phase.

The transition to the third phase, real-time phase, starts from allocating resources. During this phase, an application uses MPI/RT objects it built, and the implementation supports primary operations on the objects like channel message transfers, event deliveries, handler executions, and filling and emptying messages to and from buffers. During real-time phase an application can be in one and only one real-time mode. A real-time mode, including the first real-time mode of the real-time phase, begins when MPIRT_COMMIT is called. The real-time mode of an application is defined by the set of MPI/RT objects that th application wants to use in this mode that is passed as an input parameter to MPIRT_COMMIT function. This function allocates system resources, and MPI/RT can build a schedule for resource use and communication message transfers for that real-time mode. The MPIRT_COMMIT function fails if the QoS requirements cannot be satisfied for all application-requested objects. This process, known as the global admission test, is performed in order to guarantee the predictability of the communication and other MPI/RT operations.

MPI/RT's admission test matches the resources requested by the application to available system resources. If any of the requested objects cannot be honored, the entire test fails. This procedure inserts Operations Research and Real-Time theories results [1, 11, 16] into a middleware development in order to support implementations that build schedules for real-time resource utilization. The standard does not specify a protocol or an algorithm for instituting the admission test, which is platform hardware, software, and other middleware dependent. It is possible for an implementation to provide a means for building the real-time schedule for an application real-time mode off-line, before an application is executed, which can reduce the execution time and storage required to implement the commit function.

[4]Similar to *mpirun* of most MPI implementations.

In a real-time mode of the real-time phase, the application uses system resources such as network bandwidth and memory in a controlled and predictable manner so that the communication operations complete within the QoS requirements. If violations should occur, user-specified handlers defined for the events for these conditions, if they are defined, will fire. These handlers can make corrective action. The application can transit to another real-time mode of the real-time phase by calling MPIRT_COMMIT with a different collection of the user-requested objects (this includes the "same" objects with different quality-of-service requests). Only one real-time mode is active at a time.

An application leaves real-time phase and returns to the setup phase when MPIRT_UNCOMMIT is called. At this point resources allocated for the objects are freed, and no further MPI/RT-facilitated communications are possible until another MPIRT_COMMIT function is called that transitions an application into a real-time phase. Since an application is in setup phase, the application is allowed to create additional MPI/RT objects or modify existing ones and then re-enter real-time phase by calling MPIRT_COMMIT again. If an application does not have $\top$ (final) any further need for an MPI/RT object, an application can destroy that object by calling the $\perp$ (final) appropriate MPIRT_*name-of-class*_FREE operation.

The cleanup phase begins when the MPIRT_FINALIZE function is called. This function can only be called when the application is in the setup phase. An application should destroy all MPI/RT objects it created before entering cleanup phase. Once in this phase, the application may not call any MPI/RT functions except MPI/RT inquiry functions: MPIRT_IS_INITIALIZED and MPIRT_IS_FINALIZED. $\perp$ (final2)

1.1.5 The MPI/RT API

The MPI/RT API has been designed using object-oriented (OO) design principles and is intended to facilitate OO programming language bindings, even though the standard supports languages like `C` as well. The standard currently defines bindings for `C` and `C++`. However, it does not preclude implementations for other languages. In an OO fashion, descendent classes inherit attributes and methods from their ancestors. Non-leaf classes are pure virtual and cannot be directly instantiated by users. Additionally, MPI/RT objects are opaque (that is, their internal structures are invisible to users and must be accessed via class methods). The class hierarchy is rooted at class MPIRT_OBJECT, which contributes parameters such as name and methods for decorations (attribute caching) to all MPI/RT objects.

Parameters in MPI/RT objects are accessed via MPIRT_*name-of-class*_GET_*name-of-parameter* and MPIRT_*name-of-class*_SET_*name-of-parameter* functions. These functions store and return references to other MPI/RT objects that are constituent of the requested object. This is significant because it eliminates unnecessary copying of data, but places a greater object management burden on users, who must ensure that objects in use by the MPI/RT system are not inadvertently destroyed. Also, if any parameter values are modified using references returned by the MPIRT_*name-of-class*_GET_*name-of-parameter* function, it has the effect of modifying all objects which use that object as its constituent. Since changing parameters for committed objects is not permitted, modification of referenced parameter objects is disallowed once the committable object is committed. The parameter object may, however, be duplicated using its MPIRT_*name-of-class*_DUP function, the new identical object modified and used to set parameters in other (as yet uncommitted) objects.

MPI/RT also allows the duplication of all objects because it is a shorthand for building analogous structures, possibly sharing underlying resources (common object body). However, the newly duplicated copy of a committed object is not automatically committed. Users can modify the parameters of the new copy and must commit it before actual use.

Committable Objects

MPI/RT objects that reflect deferred early binding are descendents of the virtual class MPIRT_COMMITTABLE. Committable objects participate in the admission test performed in the MPIRT_COMMIT function. Once committed, the parameters of these objects cannot be altered until they are subsequently decommissioned by the MPIRT_UNCOMMIT function or by the next MPIRT_COMMIT in which they do not participate. Each of these objects can be thought of as having an object descriptor and object attributes, but without an object body (consisting of implicated system communication resources) until it is committed. Some examples of committable classes are MPIRT_BUFFER, MPIRT_BUFITER, MPIRT_BCAST_CHANNEL, MPIRT_PTCHANNEL, MPIRT_HANDLER, MPIRT_TRIGGER, and MPIRT_RECEPTOR. Non-committable objects, in general, are used to specify parameters for committable objects. For example, MPIRT_QOS specifies the QoS requirements for channels, handlers and event-delivery abstraction objects.

Containers

MPI/RT provides two container classes, MPIRT_CSET and MPIRT_CVECTOR_BASE, that are used to store collections of references to selected MPI/RT objects (descendents of MPIRT_OBJECT) that represent resources used in communication operations. It is erroneous for a user to build a container which includes a reference to itself (either directly or indirectly).

The MPIRT_CSET class is an abstraction of an unordered, variable-size collection of references to MPI/RT objects. Duplicate references are disallowed. Operations are provided for inserting, removing, and testing membership of elements; for querying for the number of elements; and for iterating through the elements in the set.

The MPIRT_CVECTOR_BASE class is an abstraction of an array of references to MPI/RT objects. It provides constant-time random access to indexed elements. Operations are provided for querying, assigning, and removing elements at a specified index and for resizing the vector container. This is an abstract class that has two leaf children, MPIRT_CVECTOR and MPIRT_GROUP. Both of them inherit all the container and vector container operations, but MPIRT_CVECTOR also has a create operation, while MPIRT_GROUP does not.

The container class MPIRT_GROUP is provided to represent vectors of MPIRT_TASK_ADDRESSes. This special vector container class (a subclass of MPIRT_CVECTOR_BASE) is needed because an MPIRT_TASK_ADDRESS is a special MPIRT_OBJECT that is created by an implementation, and users can only test equality and duplicate it. MPIRT_GROUP is similar in functionality to MPIRT_CVECTOR, except that gaps in the vector container are not permitted. Whenever a reference to a MPIRT_TASK_ADDRESS is removed from the container, the container's contents are shifted to eliminate the gap, and the size of the container is decremented. A special MPIRT_GROUP_WORLD is created by the MPIRT_INIT function, and all other groups can be created by a user via MPIRT_CVECTOR_BASE_DUP operations using a group as an input vector container.

Buffers

Dynamic memory allocation is a major source of unpredictability during application execution. For example, the memory required for a critical communication operation may not be available or may need to be allocated from virtual memory, taking an unpredictable amount of time. Such unpredictability makes it difficult for a system to satisfy an application's QoS requirements. Therefore, MPI/RT utilizes the concepts of buffers and buffer management. The main memory management idea in MPI/RT is to allocate all needed system resources outside of application's real-time modes.

In this way, dynamic memory allocation and the associated unpredictability during communication operations are substantially reduced.

MPI/RT's buffers are primarily used as storage for messages, but can also be used as storage in intermediate computation operations. A buffer has a data descriptor that consists of dataspec[5] and bufsize[6]; that is, it represents sufficient storage for a bufsize number of consecutive elements of the specified dataspec. The standard dataspecs supported by MPI/RT are integer, floating point, double precision, logical, character, byte, and time[7]. The MPIRT_BUFFER class provides functions for querying (get) and assigning (set) the dataspec and bufsize and for establishing a buffer object in a user-allocated memory.

The standard provides several buffer-partitioning operations that allow users to carve one large buffer into a container vector of smaller, uniform-sized buffers, preserving the original dataspec. Additionally, users can create a buffer at any location in a larger buffer, preserving the original dataspec.

Buffer Iterators

During a communication operation, MPI/RT sends data from (or receives data to) a buffer. However, the order in which the buffers are sent or filled is significant. The MPIRT_BUFITER class provides a mechanism for managing the order in which buffers are accessed. In particular, a buffer iterator attaches a policy to a container allowed_buffer_list with buffers, and this policy governs the order in which buffers are inserted and removed from the buffer iterator. Currently the standard specifies *unordered*, *LIFO*, *FIFO*, and *sorted* user-selectable policies. Several more policies specific to the vector container of buffers are also specified. These *mapped* and *ordered_mapped* policies specify that buffers can be mapped pairwise among all buffers residing in the same index in the allowed_buffer_list with *ordered* version also specifying that the buffer will always appear in the order they are specified in the allowed_buffer_list. The container with buffers initially to be placed in the buffer iterator is also specified. The standard defines accessor operations for inserting and removing a buffer from a buffer iterator, and these operations are the main run-time operations and the only operations needed for a zero-sided communication.

Figure 5.8 illustrates the state transition diagram for buffer iterators. It depicts the two-phase procedure for the creation and the movement of buffers into and out of a buffer iterator object.

Channels

Channels are unidirectional, logical conduits through which data travels from one process to another with a particular QoS. An instance of the MPIRT_PTCHANNEL class represents a point-to-point channel endpoint. Each channel endpoint object has two buffer iterators associated with it. The input buffer iterator supplies the buffers (in the user-specified order) that participate in the channel's communication operation. When a buffer is consumed (sent or filled) by the communication operation, its reference is transferred from the input buffer iterator to the output buffer iterator. It is possible for several channels to draw buffers from a shared input buffer iterator or to insert buffers into a shared output buffer iterator. However, only a single channel uses a buffer in the shared iterator. It is also possible to chain buffer iterators such that the output buffer iterator of one channel is the input buffer iterator for another.

[5] MPI uses `datatype` nomenclature for it.

[6] MPI uses `count` nomenclature for it.

[7] Unlike MPI, MPI/RT standard does not support user-defined (e.g., hierarchical) dataspecs.

Figure 9.1 illustrates the state transitions of a point-to-point channel endpoint. To send a single message, a buffer is first inserted into the input buffer iterator. Then the MPIRT_CHANNEL_START function is called to initiate the transfer. For two-sided communication all the channel endpoints must call MPIRT_CHANNEL_START for the message transfer to occur. For one-sided communication all but one channel endpoints must first transition into the MPIRT_CHANNEL_ACTIVATED state by issuing MPIRT_CHANNEL_ACTIVATE. The remaining channel endpoint may call the MPIRT_CHANNEL_START function every time it needs to transfer a single message. Since either the channel receiving endpoint(s) or the channel sending endpoint can be the designated endpoint that transfers the message, this allows MPI/RT to support both the push and pull models of one-sided communication.

For high performance systems, it is desirable to transfer multiple messages as rapidly as message buffers are filled on the sending endpoint(s) of a channel, as output message buffers are available on the receiving endpoint(s) of a channel and which the network allows, all within QoS parameters. To achieve this, the application calls the MPIRT_CHANNEL_ACTIVATE function. In this state, the channel consumes buffers as specified by the channel's QoS from the input buffer iterator and transfers them to the output buffer iterator. An activated channel remains in that state until the MPIRT_CHANNEL_DEACTIVATE function is called.

In order to set up and initiate zero-sided communication, an application creates and commits a channel with the QoS that controls the channel message transfers and calls MPIRT_CHANNEL_ACTIVATE on all of its endpoints. Under this configuration, messages are automatically transferred either at fixed time intervals for the time-driven paradigm or by event occurrences for the event-driven paradigms.

Besides point-to-point channels, which connect two processes, MPI/RT supports collective operations such as broadcast, gather, scatter, reduce and barrier that connect all processes in a group.

Event Triggers, Receptors, and Handlers

During the application's real-time mode, the application raises events explicitly by calling the MPIRT_TRIGGER_RAISE function of MPIRT_TRIGGER objects that will be delivered by an implementation to MPIRT_RECEPTOR objects that share an event_name with the MPIRT_TRIGGERs. Both MPI/RT generated (implicit) and user created (explicit) events are identified by unique event_names that couple triggers to receptors. An application registers explicit triggers to raise events by creating explicit receptors to receive events and by committing triggers and receptors respectively. A trigger has a QoS parameter describing the frequency by which an event is raised. A receptor's QoS parameter describes the requirements for the event delivery from a trigger to the receptor once an event is raised. Any number of processes within a scope of the event kind may register receptors for a single event_name. An implementation is free to create its own triggers and receptors and use standard-defined functionality to support explicit triggers and receptors and for the implementation's own use, like implementation synchronization and notification.

MPI/RT also defines events with uniquely defined event names for implicit triggers and receptors for MPI/RT object conditions such as a channel QoS error or a buffer iterator overflow. In order to receive these implicit events, users can create receptors for these implicit triggers by using the MPI/RT-defined identifying name for that implicit event. Similarly, users can create trigger objects with the same name as an MPI/RT implicit trigger. However, by improperly raising MPI/RT implicit events, users may interfere with the proper functioning of MPI/RT objects.

MPIRT_HANDLER objects specify the application-defined functions that are executed in response to receiving event notification by receptors. Handlers may be invoked synchronously or asynchronously with respect to the receptor to which they are attached. Synchronous handlers are invoked in the same context as the receptor thread, that is, as a subroutine, in the order defined ⊤ (final) by the user. Asynchronous handlers are scheduled by the receptor according to the handlers' QoS outside the context of the receptor thread prior to processing synchronous handlers. ⊥ (final)

Instrumentation

The MPI/RT system automatically maintains a number of scalar values (such as the number of messages transmitted over a channel) called metrics, which are uniquely identified by their names. Since these metrics may not be directly accessible to users, MPI/RT provides MPIRT_PROBE objects which can be attached to the metrics. A probe measures the change in a metric over time by utilizing a *mark* and *accumulate* strategy. The MPIRT_PROBE_MARK function causes the probe to record the current value of the metric. The MPIRT_PROBE_ACCUMULATE function returns the difference between the current metric value and the value recorded by MPIRT_PROBE_MARK.

Users may also create their own metrics using MPIRT_REGISTRY_ADD_METRIC in order to extend the capability of the built-in instrumentation facilities provided by an implementation.

1.2 Introduction for Users

This section describes the basic elements of an MPI/RT program. The order between the operations ⊤ (final) presented below is not required but is legitimate and typical for an application. ⊥ (final)

First, an application must call the MPIRT_INIT function before calling any other MPI/RT function. After initialization, the application prepares the objects needed for real-time communications:

Buffer objects The application creates the buffer objects. The application can either allocate the memory for them itself or can request an implementation to allocate memory at commit time by specifying MPIRT_SYSTEM_MEM_ALLOC for them.

Vector containers The application creates vector containers and fills them with buffers to be passed to buffer iterators. The application can also curve the previously created buffer(s) ⊤ (final) into smaller sized buffers that are grouped together by the implementation into container vectors.

Buffer iterator objects The buffer iterator objects are created. For each buffer iterator an allowed buffer list of previously created buffer vector containers and an initial list of buffers can be specified. The buffer iterator policy that defines the queuing order is defined. Some policies also define the matching between the allowed buffer list for a channel which will use the defined buffer iterator as input buffer iterator with the allowed buffer list for an input buffer iterator for other channel end points. ⊥ (final)

QoS objects QoS objects are created for all channels, event receptors, event triggers, and handlers. These QoS objects are used during the creation of the channels, event receptors, event triggers, and handlers.

Handler functions An application defines functions to be used by handlers.

Handler object The handler objects are created using the constituents created earlier, including QoSs and handler functions. ⊤ (final)

Event names The application creates names for event kinds for explicit events and determines names for event kinds for implicit names it would like to use.

Groups New groups are created for collective operations and event delivery abstraction objects (triggers and receptors) as needed.

Receptor objects The event receptor objects are created for specific event kinds using the constituents created earlier, including receptor QoSs, handlers, event kind names, and groups.

Trigger objects Trigger objects are created for specific event kinds using the constituents created earlier, including QoSs and event names.

Channel objects The channel objects are created using QoS, groups, and buffer iterator objects created earlier.

Commit container object The container is created, and all the committable objects (channels, triggers, handlers, receptors, buffer iterators, and buffers) that will enter real-time mode are inserted into it.

Commit The MPIRT_COMMIT function is called to perform the admission test. If it succeeds, the program enters its real-time mode of the real-time phase, and the resources are allocated for all created objects.

Once in the real-time mode, the application can activate the channels (call MPIRT_CHANNEL_ACTIVATE) or call MPIRT_CHANNEL_START to transfer one message at a time. A message is sent over a channel when the sender process inserts a buffer with a message into the channel's input buffer iterator. When the channel transfer operation completes, the buffer with a message is placed in the sending channel's output buffer iterator and the channel is available for reuse. At the receiving end of the channel, the receiver process puts a buffer into the channel's input buffer iterator. When the receive operation completes, the filled buffer is moved to the receiving channel's output buffer iterator. The insertion and removal of buffers from buffer iterators trigger implicit events that can be handled by the program in order to avoid polling loops when sending and receiving messages.

A user can change one the real-time mode for another. In order to change the real-time mode, the user need to create a container with all the objects that need to be committed for a new real-time mode. When the operations on currently committed objects complete (that means the the channels should be in the MPIRT_CHANNEL_IDLE state), the new commit operation is called. All the previously committed objects that appear in the container with the new objects to be committed remain committed and retain their state over the commit operation. When the commit returns successfully the application enters the new real-time mode. Otherwise it remains in the previous real-time mode.

To exit the real-time phase, in the real-time mode the application deactivates the channels (calls MPIRT_CHANNEL_DEACTIVATE) and calls MPIRT_UNCOMMIT in order to free all resources the real-time mode may have allocated for the objects. The application can then destroy the objects it had created before. Finally, before it exits, the application must call the MPIRT_FINALIZE function.

1.3 Introduction for Implementors

MPI/RT communication is an architecture-independent communications paradigm. The API can accommodate various memory models, dedicated hardware, and specialized software in both hard and soft real-time environments. This real-time capability also takes advantage of architecture-independent, portable features in real-time applications and hardware to achieve a portable API for real-time programming. This section discusses MPI/RT functionality and provides an overview of issues for implementors.

1.3.1 The Basics

As has been mentioned, MPI/RT has been designed from an object-oriented standpoint. This does not, however, preclude implementation in non-object-oriented languages. Implementing the object hierarchy in C, for instance, should be feasible using structures, unions, and other techniques common to previous object-oriented works.

Also in line with other works, developers may find it advantageous to define the specified MPI/RT objects as pointers or "smart-pointers," depending on the capabilities of the implementation language, to minimize function parameter passing costs, to reduce or eliminate scoped-definition problems, and to implement reference counting.

Inheritance and polymorphism, which are common throughout the design of this specification and not present in most non-object-oriented languages, are represented here in the C bindings for the functions as individual functions that support the operation with the name of the appropriate owning object in the function name and the appropriate parameter in the parameter list. This eliminates warnings and errors that would otherwise be generated by strongly-typed languages. This does increase the function count but with the realization that many of these functions are similar, and some implementation short-cuts can be taken to reduce the amount of work required by the implementor.

Care has been taken to minimize the penalties of implicit copies in languages such as `C++`. Reference semantics, with and without `const` directives, are used whenever appropriate to eliminate these hidden dangers.

In keeping with minimizing hidden dangers, throughout this design all copy functions are considered to be *shallow*. This minimizes the overhead of copying objects and the danger of *deep* copy resource allocation, but does not prevent the ability to construct any desired deep copy behavior. Users can perform their own deep copies by writing functions that recursively use the shallow copy functions or use libraries written by others. These deep copy functions are considered to be convenience functions and were not included in this specification because libraries can be portably written by developers using the MPI/RT API and third parties to achieve this exact desired behavior. In addition, there may be varieties of specific and differing copy semantics desired that cannot be predicted by this forum or encompassed easily in a single or a few copy functions.

1.3.2 The Admission Test

The key component of MPI/RT is the ability of the user to inform the library implementation of the requirements for correct behavior of his application by building up objects that encapsulate the performance and resource requirements of memory, timing, communication patterns and demands, and performance parameters. These requirements are built up by constructing various objects to describe how the resources are to be used and then using them as building blocks for other objects to describe their behavior. For example, buffer objects describe memory resources, and they are building blocks of buffer iterator objects which describe queuing methodology which, along with appropriate quality-of-service objects which describe performance characteristics, are building blocks for channels. All of this information informs the implementation of the performance and resource requirements for the message channel, how to store the data as it arrives or is sent, and the location and size of each data block.

All objects, such as triggers, receptors, and handlers, that describe the performance and resource requirements of the program are submitted by the user to an admission test. This admission test may be performed at runtime by the commit function or may be an off-line procedure. In either case, the admission test confirms that the implementation, and thereby the target hardware, is able to satisfy the user-requirements for the program[8]. Since the admission test depends on information given to it by the user, a successful admission test indicates that the target platform can give the user the required resources and performance based on the information given. It is the responsibility of the user to provide correct information. This information can be obtained by the user by first giving very liberal requirements and using instrumentation in a repetitive process to tighten requirements so that the platform can give better results, or it can be a known quantity, such as a minimum frame rate, that must be obtained. In the first case, the program is adapted to the platform itself. In the latter case, the platform is required to meet some set requirements. It may be the case that the platform is unable to meet the user requirements, in which case the admission test should fail.

While much of the code to implement the MPI/RT specification should be rather straightforward to implement, the admission test is the most complex portion of the specification. There is a wide variety of known techniques that can be used to generate schedules for a single process and some that can be used to generate schedules of multiple processes that run on multiple processors. Some of these techniques are suitable for runtime generation, but others are more suited for off-line use. Indeed, the computational power of the platform itself may not be enough for a runtime admission test.

Adding to the complexity of the admission test is that, depending on the quality of the implementation, a different admission test will be required for each platform. Even small changes to a platform, such as increasing the clock speed of the processor and all other things being equal, may require different admission tests. Simulators may help model heterogeneous systems, and parameterized admission tests may also be a solution to target a family of platforms.

1.3.3 Erroneous Programs and Debugging Libraries

Erroneous programs are those that do not meet the semantic requirements of the MPI/RT specification. Such programs may result in deadlocks or crash the system. For example programs that pass incorrect or invalid parameters to MPI/RT functions and programs that cause race conditions are erroneous. Furthermore, due to the unpredictable nature of race conditions, these programs may behave correctly some of the time. Semantic errors can be difficult to detect during runtime and the execution of error detection code can negatively impact runtime performance.

In order to improve runtime performance while providing debugging support to users, an implementation may consist of two kinds of libraries as follows:

- Debugging — This library will perform extensive runtime error and parameter consistency checking, possibly at the cost of performance.

- Release — This library will perform a reduced set of error and parameter consistency checks in order to improve runtime performance.

The doocument lists the possible error codes returned by each operation. Following is a list of generic error codes that are returnd when icorrect or invalid parameters are passed to

[8]It is of note that this admission test need not be performed at runtime. Off-line admission tests can be performed, and the commit function can load stored schedules, for instance. It is also of note that the schedule for one process necessarily depends on the schedules of other processes in the MPI/RT job because the job is a parallel program. Because of this, admission tests may be complex programs or subroutines that may not be feasible for runtime invocation.

MPI/RT operations: MPIRT_ERR_INV_BUFFER, MPIRT_ERR_INV_BUFITER, MPIRT_ERR_INV_CHANNEL, MPIRT_ERR_INV_CONTAINER, MPIRT_ERR_INV_CSET, MPIRT_ERR_INV_CVECTOR, MPIRT_ERR_INV_GROUP, MPIRT_ERR_INV_HANDLER, MPIRT_ERR_INV_KEYVAL, MPIRT_ERR_INV_METRIC, MPIRT_ERR_INV_NAME, MPIRT_ERR_INV_OBJECT, MPIRT_ERR_INV_PARAM, MPIRT_ERR_INV_POLICY, MPIRT_ERR_INV_PROBE, MPIRT_ERR_INV_QOS, and MPIRT_ERR_INV_RANGE.

The exact extent of the checks performed by each kind of library is left up to the implementers. Because of the extra error checking performed by the debugging library, it is possible for a MPI/RT program using the debug library fails the admission test, but passes the admission test using the release library. ⊥ (final)

1.3.4 Conclusion

In closing, implementors are encouraged to "do the right thing" and create high-quality MPI/RT libraries. Users desire high degrees of available internal instrumentation, good documentation, and high-performance, low-overhead libraries. Implementors should provide detailed performance information such as function call timings and response times of events to the users of their systems.

Tool development is also encouraged. Simulators, off-line and runtime admission tests, and application construction tools are always desired and are marketable items.

Part I

Concepts and Basic Objects

Chapter 2

MPI/RT Objects

Current Status: First formal vote on sections 2.1, passed 8/0/0 on 2/98.
First formal vote on section 2.2 passed 7/0/0 on 4/98.
First formal vote on section 2.3 passed 11/0/0 on 6/98.
First formal vote on section 2.4 passed 11/0/0 on 6/98.
First formal vote on section 2.5 passed 11/0/0 on 6/98.
First formal vote on section 2.6 passed 9/0/2 on 6/98.
Second formal vote passed 9/0/2 on 7/23/98.
Third formal vote passed 9/0/2 on 12/3/98.

2.1 Overview

The MPI/RT API is based on object-oriented methodology. Objects, methods, and copy semantics are defined for a number of objects, pertaining to data transfer, quality-of-service (QoS) management, and other aspects of both declaration and operation invocation.

MPI/RT objects are **opaque**, their shapes and sizes are not visible to the user, and they are not directly accessible to the user. MPI/RT also defines certain non-opaque types that are directly accessible to the user. A subset of these is transmittable. MPI/RT objects consist of an **object descriptor**, the **object body**, and any other information that an implementation stores with an object for its own use, which is invisible to the user. Some object types do not have an object body. The object descriptor defines the persistent attributes of an object, such as size or type, whereas the object body contains the resources. MPI/RT supports object decoration.

MPI/RT manages system memory that is used for storing internal representations of various MPI/RT objects such as groups, point-to-point channels, collective channels, buffer iterators, buffers, handlers, and event triggers and receptors.

For each object class, *name-of-class*, there is a type, MPIRT_*name-of-class*_TYPE, which uniquely defines this class (See Chapter 3).

Each MPI/RT object has a standard set of operations defined on it. For each object, there is an appropriate MPIRT_*name-of-class*_CREATE for the creation of an instance of the object which appears in the chapter of this document where the object is defined. Also, for all MPI/RT objects, the functions MPIRT_OBJECT_DUP for creating a copy of an object instance and MPIRT_OBJECT_FREE for deleting an object instance are defined. MPI/RT defines the constant

MPIRT_*name-of-class*_NULL, which is returned in the INOUT parameter for the object reference for MPIRT_OBJECT_FREE operations. These operations are common for all MPI/RT objects and are defined in section 2.2.

The MPI/RT objects also share several operations on an object descriptor that are defined in section 2.2: MPIRT_*name-of-class*_GET_*name-of-parameter* for query of each object descriptor parameter,
and MPIRT_*name-of-class*_SET_*name-of-parameter* for setting each object descriptor parameter. All MPIRT_*name-of-class*_GET_*name-of-parameter* operations return references to the original objects to the caller. A call to MPIRT_OBJECT_FREE on any reference to a non-committed MPI/RT object destroys that object. MPI/RT defines *containers* in order to pass parameters that are not a single entity. In order to minimize the system's use of dynamic memory allocation, which in turn reduces the predictability and timing determinism of a program, the MPI/RT API chooses to work with by-reference parameter passing of query-operation information.

Object attributes are supported. These permit objects to be "decorated" in the object-oriented sense. Arbitrary parameters can be added to objects, with appropriate callback functions defined whenever an object is to be duplicated or deleted. In this way, the objects become simply extensible by the application.

Two objects defined in MPI/RT: MPIRT_CLOCK and MPIRT_REGISTRY are singletons and do not support the generic object operations operations. An implementation creates these objects at MPIRT_INIT and destroys (frees) them at MPIRT_FINALIZE. These objects do not support duplication, equality testing, object decoration, as well as get and set operations for the name, because they do not have the name parameter.

Certain objects have the property that both users and implementors can allocate resources for their object bodies. An example of these objects is the buffer objects, instances of which may be built by the user or else deferred for system declaration using special memory.

MPI/RT defines several objects that are used to support communication, memory management, and event processing: buffers, buffer iterators, point-to-point channels, collective channels, handlers, groups, and event triggers and receptors. All objects are persistent in MPI/RT, meaning that there is a planned transfer paradigm that is set up once and used repeatedly in order to exploit program temporal locality.

Certain objects are said to be *committable,* meaning that they undergo two-phase initialization, a mechanism by which MPI/RT achieves "deferred early binding." For these objects, the MPIRT_*name-of-class*_CREATE operation does not create the object body; the MPIRT_COMMIT operation does this later. For the other objects, like quality-of-service objects, *create* is the only initialization operation because these objects do not have a body.

Committable objects undergo two-phase initialization. The first phase (MPIRT_*name-of-class*_CREATE) defines the interaction between objects and defines the user requirements for the object via object descriptors. The second phase (MPIRT_COMMIT) commits resources and creates the object body (*e.g.*, network and memory) and plans operations (creates schedules) based on the first phase of the initialization. The object descriptor allows a layered approach for creating and managing requested object information separately from the object body subsequently created at the commit time. The following object types are committable (see Figure 2.1): buffer, buffer iterator, handler, point-to-point channel, collective channel, trigger and receptor.

The duplicate operation of a committable object creates a new object with the same parameters specification as the original object. Using set operations, a user can request different constituted objects to be used for the new object or change any object descriptor parameter. The commit operation generates the object body for the new object by using the same constituent objects as

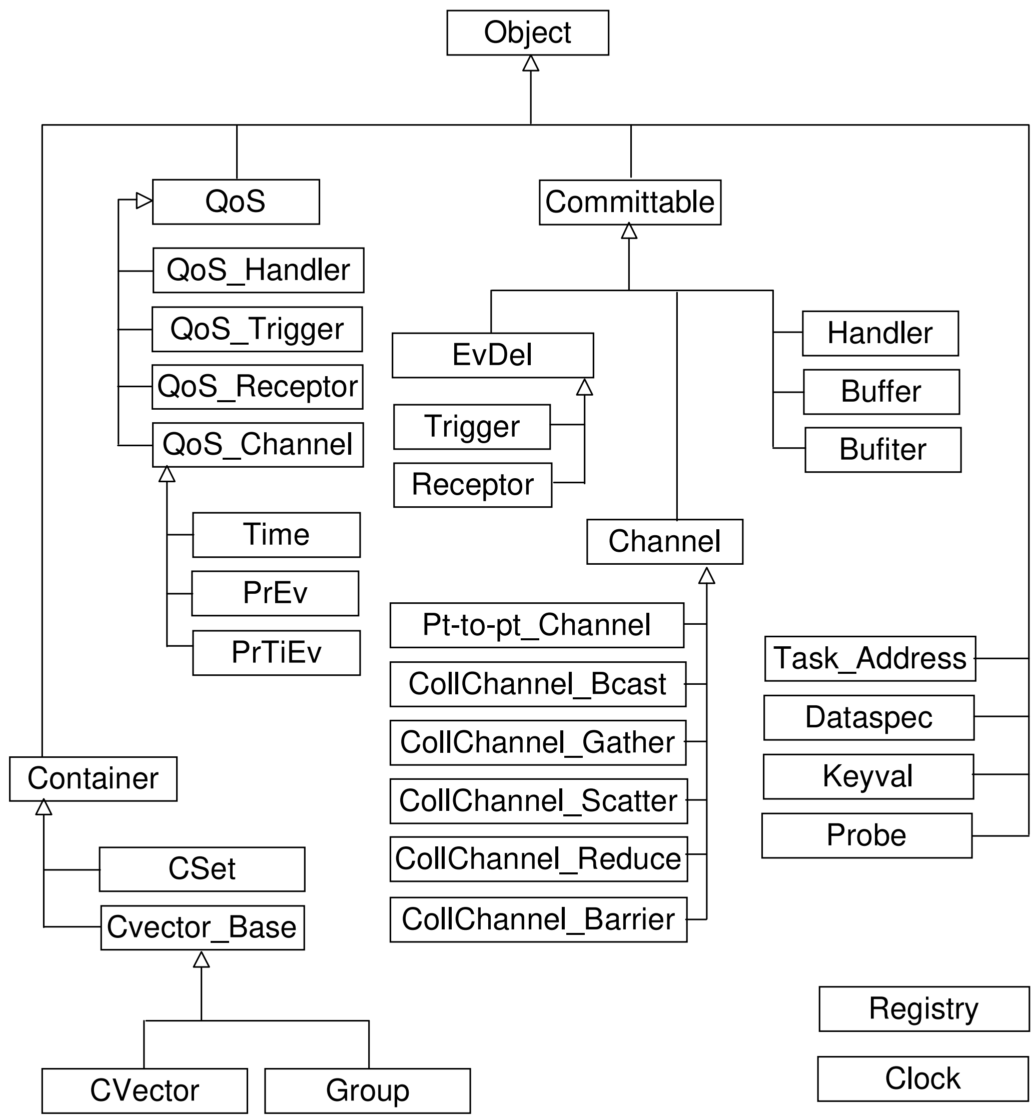

Figure 2.1: A UML-type hierarchy of the classes in MPI/RT accessible to users as part of the application programmer interface. This is primarily an `IS_A` hierarchy, in which an arrow points from subclass to superclass, and all non-leaf classes are "pure virtual" and hence are never instantiated. Methods (interfaces) and attributes are inherited from superclass to subclass.

the original one, unless the set operations were used to request new ones.

> *Advice to users.* It should be noted that a duplicated object is useless in most cases without a subsequent set[1]. This is needed to make the object distinct in some way. By way of distinction, a duplicated object that has no differences from its antecedent merely uses additional resources, and this might be of value in certain layered libraries, even without a subsequent set. (*End of advice to users.*)

The MPIRT_*name-of-class*_SET_*name-of-parameter* operations on a committed object return an error MPIRT_ERR_COMMITTED_OBJECT. Analogously, operations on the object body before it is committed also return an error MPIRT_ERR_OBJECT_NOT_COMMITTED.

Each MPI/RT object class has its own set of operations that manipulate the object body but not its descriptor.

Each object type has a state transition diagram defined for it. Most objects have a trivial state transition diagram where the object moves from nothing to uncommitted state via create or dup operations, from uncommitted to committed via commit operation (for committable objects), and back via uncommit (for committable objects) and free operations, respectively. The transition from one state to another generates an object condition that is defined in the chapter where the object creation operation is defined. The conditions in turn raise events that are delivered from the trigger (implicit) to event receptors. Receptors can have handlers attached to them that will be executed upon event arrivals (see Sections 4.2 and 4.3). The transition from one object state to another is a response of some object operation.

2.2 Generic Operations Defined on All MPI/RT Objects

This section presents some of the operations that are common for all MPI/RT objects. The operations that are common to all MPI/RT objects that deal with object decoration are presented in Section 2.3.

The following operation is a generic operation that creates a copy of an MPI/RT object. These operations work analogously for each class[2].

MPIRT_*name-of-class*_DUP (object, new_object)

IN	object	existing MPI/RT object (MPIRT_*name-of-class*)
OUT	new_object	new MPI/RT object (ref. to MPIRT_*name-of-class*)

```
int MPIRT_name-of-class_dup (MPIRT_name-of-class object,
            MPIRT_name-of-class *new_object)
```

```
MPIRT::name-of-class MPIRT::name-of-class::Operator=(const MPIRT::name-of-class
            &object)
```

```
MPIRT::name-of-class MPIRT::name-of-class (const MPIRT::name-of-class &object)
```

The duplication operation retains the object linkages to the original object as well as parametric specifications of the original object, while creating a new resource requirement for the object. The

[1] For instance, a subsequent set operations is not required for buffer objects.

[2] following the approach of the orthodox canonical form for the C++ binding, per Coplien [2].

operation does not create the object body for the new object, which is equivalent to the object's underlying resources. The set operations on the new object do not have any effect on the original object nor the original object's underlying resources.

Advice to implementors. The MPIRT_*name-of-class*_DUP function is defined equivalently to the result of this `C` function:

```
typedef union
{
   MPIRT_CHANNEL c;
   MPIRT_BUFITER bi;
   MPIRT_BUFFER  bu;
   /* etc for each MPIRT object type */
} INTERNAL_OBJECT_UNION;

typedef struct
{
   INTERNAL_OBJECT_UNION u;
   int type_discriminator;
   int refcount;
   int object_state;
   /* plus there are possibly many system/implementation
      things in here and even for each individual object
      type as well */
} MPIRT_OBJECT, *MPIRT_Object;

int MPIRT_Object_Dup(MPIRT_Object oldobject, MPIRT_Object *newobject)
{
int status = MPIRT_SUCCESS;
MPIRT_Object no;

if((no = (MPIRT_OBJECT *) malloc(sizeof(MPIRT_OBJECT))) == 0)
        {
     status = MPIRT_ERROR;
     goto failure;
        } /* allocate shell */
*no = *oldobject; /* copies everything */
no->refcount = 1; /* adjust new object */
no->object_state = UNCOMMITTED; /* system defined state */
/* any other specific system/implementation things to
   make this a 'new' object... */
        /* keeps object descriptor and object body from original,
   reinitializes the rest */
*newobject = no;
/* do something specific to do attribute copying */
status = do_attribute_cache_dup(&oldobject,newobject);
```

```
failure:
return status;
}
```

(*End of advice to implementors.*)

Advice to implementors. Because C++ does not permit inheritance of constructors, copy constructors, or assignment operators, each class must have its own specific variants of these operations. (*End of advice to implementors.*)

The following operation frees the specified object, including any system-allocated resources, and returns MPIRT_OBJECT_NULL. A call to MPIRT_OBJECT_FREE on a committed MPI/RT object returns an error: MPIRT_ERR_COMMITTED_OBJECT.

MPIRT_OBJECT_FREE (object)

INOUT	object	MPI/RT object (ref. to MPIRT_OBJECT)

`int MPIRT_Object_free (MPIRT_Object *object)`

`virtual MPIRT::Object::~Object(void)`

This object deletes all attributes attached to the object in random order and then deletes the object itself. For each attribute, the delete callback function is called (see below).

The next operation returns the object type of the specified object.

MPIRT_OBJECT_GET_OBJECT_TYPE (object, type)

IN	object	MPI/RT object (MPIRT_OBJECT)
OUT	type	type of the MPI/RT object (ref. to MPIRT_OBJECT_TYPE)

`int MPIRT_Object_get_object_type (MPIRT_Object object, MPIRT_Object_type type)`

`int MPIRT::`*name-of-class*`::Get_Object_Type (MPIRT::Object_type &type) const`

⊤ (final2) All MPI/RT objects have a name of type MPIRT_STRING_NAME. The object name must be unique among all MPI/RT objects created by this process if a user is going to create explicit or implicit receptors for the conditions of the object (see chapter 4). Hence, object names are used to generate unique event names (see Section 4.2.1), any object that may generate events of interest to the programmer must have a non-NULL, non-zero-length string for a name.

⊥ (final2) The next operation retrieves the object name. Multiple objects that don't require names for programmatic purposes may share the degenerate name MPIRT_STRING_NAME_NULL.

MPIRT_OBJECT_GET_NAME (object,name)

IN	object	MPI/RT object (MPIRT_OBJECT)
OUT	name	name of object (ref. to MPIRT_STRING_NAME)

`int MPIRT_Object_get_name (MPIRT_Object object, MPIRT_String_name *name)`

```
int MPIRT::Object::Get_Name (MPIRT::String_Name &name) const
```

The next operation assigns an object name.

MPIRT_OBJECT_SET_NAME (object,name)

INOUT	object	MPI/RT object (ref. to MPIRT_OBJECT)
IN	name	name of object (MPIRT_STRING_NAME)

```
int MPIRT_Object_set_name (MPIRT_Object *object, MPIRT_String_name name)
```

```
int MPIRT::Object::Set_Name (const MPIRT::String_Name name)
```

⊤ (final)

The following function tests equality of two like-typed objects:

MPIRT_OBJECT_IS_EQUAL (obj1, obj2, result)

IN	obj1	left MPI/RT object (MPIRT_OBJECT)
IN	obj2	right MPI/RT object (MPIRT_OBJECT)
OUT	result	outcome of comparison (enumeration)

```
int MPIRT_Object_is_equal (MPIRT_Object obj1,MPIRT_Object obj2, int *result)
```

```
int MPIRT::Object::Is_equal (const MPIRT::Object &obj2, int &result) const
```

The outcomes reported in the *result* parameter are as follows:

1. MPIRT_IS_EQUAL, if the objects are identical (the same object),
2. MPIRT_UNEQUAL, if the objects are not identical (not the same object).

⊥ (final)

2.3 Attributes: Object Decoration

MPI/RT provides a "decoration" facility that allows an application to attach arbitrary pieces of information, called *attributes*, to all objects. More precisely, the decoration facility allows a portable library to do the following:

- pass information between calls by associating it with a common key to one or more MPI/RT objects, and
- easily retrieve that information.

Attributes are attached to any MPI/RT object (groups and channels are especially good candidates). Attributes are local to the process and specific to the object to which they are attached. Attributes are not propagated by MPI/RT from one object to another except when the object is duplicated using MPIRT_OBJECT_DUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

Attributes are scalar values, equal in size to or larger than a C (respectively, C++) language pointer.

The decoration interface defined here reflects the design feature that states that attributes are stored opaquely within an object by MPI/RT. Accessor functions allow users to do the following:

- obtain a key value (used to identify an attribute across one or more objects in a process); the user specifies "callback" functions by which MPI/RT informs the application when the object is destroyed or copied, and
- store and retrieve the value of an attribute.

Decoration and callback functions are only called synchronously in response to explicit application copies or deletions of the objects to which the attributes are attached. The purpose of this callback is to permit attributes to support either deep or shallow copy semantics and to support reference counting of such information, if desired, under user control and on an attribute-by-attribute basis.

The choice of key values is under the control of MPI/RT. This allows MPI/RT to optimize its implementation of attribute sets. It also prevents conflicts between independent modules decorating information on the same objects.

MPI/RT provides the following services related to decoration. They are all local to a process.

MPIRT_KEYVAL_CREATE (name, dup_fn, delete_fn, extra_state, keyval)

IN	name	name of the key value (MPIRT_STRING_NAME)
IN	dup_fn	copy callback function for keyval (MPIRT_KEYVAL_DUP_FN)
IN	delete_fn	delete callback function for keyval (MPIRT_KEYVAL_DELETE_FN)
IN	extra_state	extra state for callback functions (MPIRT_ADDRESS)
OUT	keyval	key value for future access (ref. to MPIRT_KEYVAL)

```
int MPIRT_Keyval_create (MPIRT_String_name name, MPIRT_Keyval_dup_fn dup_fn,
             MPIRT_Keyval_delete_fn delete_fn, MPIRT_Address extra_state,
             MPIRT_Keyval *keyval)
```

```
MPIRT::Keyval (const MPIRT::String_name name, const MPIRT::Keyval::Dup_Fn dup_fn,
             const MPIRT::Keyval::Delete_Fn delete_fn, const
             MPIRT::Address extra_state)
```

MPIRT_KEYVAL_CREATE generates a new attribute key. Keys are process unique, with no assigned meaning across processes[3], and opaque to the user, though they are uniquely commensurable to integers (accessible for use in creating additional user structures). Once allocated, the key value can be used to associate attributes and access them on any locally-defined objects.

The dup_fn function is invoked when an object is duplicated (by its MPIRT_*name-of-class*_DUP function). The following is a language-independent specification of what a user keyval dup function should be. It is not a function defined in MPI/RT. The function dup_fn should be of type MPIRT_KEYVAL_DUP_FN, whose prototype is defined as follows:

[3]This follows the approach of MPI-1.

MPIRT_KEYVAL_DUP_FN (old_obj, keyval, extra_state, attribute_val_in, attribute_val_out, flag)

IN	old_obj	the object being duplicated
IN	keyval	the keyval of the attribute being duplicated
IN	extra_state	the extra_state of the keyval
IN	attribute_val_in	the value of the attribute to be duplicated
OUT	attribute_val_out	the value of the duplicated attribute
OUT	flag	indicates whether or not attribute is to be duplicated

```
typedef int (*MPIRT_Keyval_dup_fn) (MPIRT_Object old_obj, MPIRT_Keyval keyval,
              MPIRT_ADDRESS extra_state, MPIRT_Address attribute_val_in,
              MPIRT_Address *attribute_val_out, int *flag);
```

```
typedef int (*MPIRT::Keyval::Dup_Fn) (const MPIRT::Object old_obj, const
              MPIRT::Keyval keyval, const MPIRT::Address extra_state, const
              MPIRT::Address attribute_val_in, MPIRT::Address &attribute_val_out,
              int &flag);
```

The copy callback function is invoked for each key value in old_obj in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns `flag = 0`, then the attribute is to be omitted in the duplicated object. Otherwise (`flag = 1`), the new attribute value is set via attribute_val_out. The function returns MPIRT_SUCCESS on success and an error code on failure (in which case MPIRT_*name-of-class*_DUP will fail).

The dup_fn may be specified as MPIRT_KEYVAL_DUP_DEFLT_FN from both C and C++, in which case no copy callback occurs for keyval; MPIRT_KEYVAL_DUP_DEFLT_FN is a function that does nothing other than return `flag = 0`.

As a further convenience, MPIRT_KEYVAL_DUP_FN_SHALLOW is a simple copy callback; it sets `flag = 1` and returns the value of attribute_val_in in attribute_val_out.

> *Advice to users.* A valid copy (dup) function is one that completely duplicates the information by making a full duplicate copy of the data structures implicated by an attribute; another copy function might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to old_obj only).
>
> It should be noted that, since MPIRT_KEYVAL is an MPI/RT object, duplication of this object is supported. Such duplication retains the settings from the original object, but introduces a new, unique keyval so that the user can easily establish multiple attributes with shared or similar callbacks. (*End of advice to users.*)

Analogous to dup_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when an object is deleted by MPIRT_*name-of-class*_FREE or when a call is made explicitly to MPIRT_KEYVAL_DELETE_ATTR. The following is a language-independent specification of what a user keyval delete function should be. It is not a function defined in MPI/RT. The delete_fn should be of type MPIRT_KEYVAL_DELETE_FN, whose prototype is defined as follows:

MPIRT_KEYVAL_DELETE_FN (obj, keyval, attribute_val, extra_state)

IN	obj	the object for whom the attribute is being deleted
IN	keyval	the keyval of the attribute being deleted
IN	attribute_val	the value of the attribute being deleted
IN	extra_state	the extra_state of the keyval

```
typedef int (*MPIRT_Keyval_delete_fn) (MPIRT_Object obj, MPIRT_Keyval keyval,
             MPIRT_Address attribute_val, MPIRT_Address extra_state);
```

```
typedef int (*MPIRT::Keyval::Delete_Fn) (const MPIRT::Object obj, const
             MPIRT::Keyval keyval, const MPIRT::Address attribute_val, const
             MPIRT::Address extra_state);
```

This function is called by MPIRT_OBJECT_FREE and MPIRT_OBJECT_DELETE_ATTR to do whatever is needed to remove an attribute. It may be specified as the null function pointer in `C` or as MPIRT_KEYVAL_DELETE_FN_EMPTY from both `C` and `C++`, in which case no delete callback occurs for keyval.

The special key value MPIRT_KEYVAL_INVALID is never returned by MPIRT_KEYVAL_CREATE. Therefore, it can be used for static initialization of key values.

The free, dup, get/set name and equality testing operations for MPIRT_KEYVAL are inherited from MPIRT_OBJECT (See Section 2.2). Note that it is not erroneous to free an attribute key that is in use because the actual *free* operation does not transpire until after all references (in other objects in the process) to the key have been freed. These references need to be explicitly freed by the program, either via calls to MPIRT_KEYVAL_DELETE_ATTR that free one attribute instance, or by calls to MPIRT_OBJECT_FREE that free all attribute instances associated with the freed object.

2.3.1 Keyval Object Parameter Accessors

MPIRT_KEYVAL_GET_DUP_FN (keyval, dup_fn)

IN	keyval	keyval object (MPIRT_KEYVAL)
OUT	dup_fn	copy function (ref. to MPIRT_KEYVAL_DUP_FN)

```
int MPIRT_Keyval_get_dup_fn (MPIRT_Keyval keyval, MPIRT_Keyval_dup_fn *dup_fn)
```

```
int MPIRT::Keyval::Get_Dup_Fn (MPIRT::Keyval::Dup_Fn &dup_fn) const
```

MPIRT_KEYVAL_GET_DELETE_FN(keyval, delete_fn)

IN	keyval	keyval object (MPIRT_KEYVAL)
OUT	delete_fn	delete function (ref. to MPIRT_KEYVAL_DELETE_FN)

```
int MPIRT_Keyval_get_delete_fn (MPIRT_Keyval keyval,
             MPIRT_Keyval_delete_fn *delete_fn)
```

```
int MPIRT::Keyval::Get_Delete_Fn (MPIRT::Keyval::Delete_Fn &delete_fn) const
```

MPIRT_KEYVAL_GET_EXTRA_STATE (keyval, extra_state)

IN	keyval	keyval object (MPIRT_KEYVAL)
OUT	extra_state	extra state reference (ref. to MPIRT_ADDRESS)

```
int MPIRT_Keyval_get_extra_state (MPIRT_Keyval keyval, MPIRT_Address *extra_state)
```

```
int MPIRT::Keyval::Get_Extra_State (MPIRT::Address &extra_state) const
```

MPIRT_KEYVAL_SET_DUP_FN (keyval, dup_fn)

INOUT	keyval	keyval object (ref. to MPIRT_KEYVAL)
IN	dup_fn	duplication function (MPIRT_KEYVAL_DUP_FN)

```
int MPIRT_Keyval_set_dup_fn (MPIRT_Keyval *keyval, MPIRT_Keyval_dup_fn dup_fn)
```

```
int MPIRT::Keyval::Set_Dup_Fn (MPIRT::Keyval::Dup_Fn dup_fn)
```

MPIRT_KEYVAL_SET_DELETE_FN (keyval, delete_fn)

INOUT	keyval	keyval object (ref. to MPIRT_KEYVAL)
IN	delete_fn	delete function (MPIRT_KEYVAL_DELETE_FN)

```
int MPIRT_Keyval_set_delete_fn (MPIRT_Keyval *keyval,
              MPIRT_Keyval_delete_fn delete_fn)
```

```
int MPIRT::Keyval::Set_Delete_Fn (MPIRT::Keyval::Delete_Fn delete_fn)
```

MPIRT_KEYVAL_SET_EXTRA_STATE (keyval, extra_state)

INOUT	keyval	keyval object (ref. to MPIRT_KEYVAL)
IN	extra_state	extra state reference (MPIRT_ADDRESS)

```
int MPIRT_Keyval_set_extra_state (MPIRT_Keyval *keyval, MPIRT_Address extra_state)
```

```
int MPIRT::Keyval::Set_Extra_State (const MPIRT::Address extra_state)
```

2.3.2 Object Attribute Manipulation Functions

MPIRT_OBJECT_ATTACH_ATTR (object, keyval, attribute_val)

INOUT	object	object to which attribute will be attached (ref. to MPIRT_OBJECT)
IN	keyval	key value, as returned by MPIRT_KEYVAL_CREATE (MPIRT_KEYVAL)
IN	attribute_val	attribute value (MPIRT_ADDRESS)

```
int MPIRT_Object_attach_attr (MPIRT_Object *object, MPIRT_keyval keyval,
              MPIRT_Address attribute_val)
```

```
int MPIRT::Object::Attach_Attr (const MPIRT::Keyval &keyval, const
              MPIRT::Address attribute_val)
```

This function stores the specified attribute value attribute_val for subsequent retrieval by MPIRT_KEYVAL_RETRIEVE_ATTR. If the value is already present, then the outcome of the function becomes as if MPIRT_KEYVAL_DELETE_ATTR was first called to delete the previous value (and the callback function delete_fn was executed), and a new value was next stored. The call is erroneous if there is no object keyval; in particular MPIRT_KEYVAL_INVALID is an erroneous key value.

It is illegal to attach an attribute to the object MPIRT_KEYVAL.

MPIRT_OBJECT_RETRIEVE_ATTR (object, keyval, attribute_val, flag)

IN	object	object to which the attribute is attached (MPIRT_OBJECT)
IN	keyval	key value (MPIRT_KEYVAL)
OUT	attribute_val	attribute value, unless flag = MPIRT_FALSE (ref. to MPIRT_ADDRESS)
OUT	flag	MPIRT_TRUE if an attribute value was extracted; MPIRT_FALSE if no attribute is associated with the key (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Object_retrieve_attr (MPIRT_Object object, MPIRT_keyval keyval,
              MPIRT_Address *attribute_val, MPIRT_Boolean *flag)
```

```
int MPIRT::Object::Retrieve_Attr (const MPIRT::Keyval &keyval,
              MPIRT::Address &attribute_val, MPIRT::Boolean &flag) const
```

This function retrieves the attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on `object` for that key; in such case, the call returns `flag = MPIRT_FALSE`. In particular, MPIRT_KEYVAL_INVALID is an erroneous key value.

MPIRT_OBJECT_DELETE_ATTR (object, keyval)

INOUT	object	object to which the attribute is attached (ref. to MPIRT_OBJECT)
IN	keyval	key value of the deleted attribute (MPIRT_KEYVAL)

```
int MPIRT_Object_delete_attr (MPIRT_Object *object, MPIRT_keyval keyval)
```

```
int MPIRT::Object::Delete_Attr (const MPIRT::Keyval &keyval)
```

This function deletes the attribute from the object decoration collection by its key. The function invokes the attribute's delete function delete_fn specified when the keyval was created.

Whenever an object is duplicated using the function MPIRT_OBJECT_DUP, all call-back copy functions for attributes that are then currently set are invoked (in arbitrary order). Whenever an object is deleted using the function MPIRT_OBJECT_FREE, all callback delete functions for attributes that are currently set are invoked.

2.4 Containers

MPI/RT defines containers for certain collections of basic objects. Providing an abstraction for arrays is also consistent with modern object-based and object-oriented programming. The types of containers supported by MPI/RT are *sets* and *vectors*. Set containers are denoted by *S-container* and vector containers by *V-container*.

Containers provide an aggregation abstraction for MPI/RT. They are MPI/RT objects and are constructed and destructed via generic MPI/RT operations defined in Section 2.2. Additional operations on containers that deal with manipulating containers' members are defined below.

> *Advice to implementors.* No explicit details or requirements for how containers in MPI/RT should be implemented are given here. However, *V-containers* should support $O(1)$ time for access to elements. (*End of advice to implementors.*)

Both *V-containers* and *S-containers* have the following properties:

- Containers hold references to zero or more MPI/RT objects, not copies.
- Containers are resizeable. $\top$ (final)
- Containers are MPI/RT objects, and hence containers can hold references to other containers. Container's references can not form a loop. That means that a container can not hold a reference to itself nor can it hold a reference to a container which directly or indirectly (transitively) points to it. $\bot$ (final)

Vector containers have the following additional properties:

- They define an index order, $0, \ldots, N-1$, for N entries.
- They provide random access to any entry in $O(1)$ time.

Set containers have the following additional properties:

- Set container members do not have any order. $\top$ (final)
- There are no identical members (duplication) in a set container. $\bot$ (final)

Set containers are denoted MPIRT_CSET, whereas vector containers are denoted MPIRT_CVECTOR_BASE. MPIRT_CVECTOR_BASE is a virtual class with two children, MPIRT_CVECTOR and MPIRT_GROUP, that are leaves. This standard defines the creation operation for MPIRT_CVECTOR and MPIRT_GROUP, but not for MPIRT_CVECTOR_BASE. Both MPIRT_CVECTOR and MPIRT_GROUP inherit operations defined for MPIRT_CVECTOR_BASE.

2.4.1 Container Constructors

The following functions define mechanisms for creating containers. Standard operations on MPI/RT objects are applicable to containers (see Section 2.2). These include copy-constructor, freeing of a container, testing containers equality, and getting and setting a container's name. The operations MPIRT_CSET_FREE, MPIRT_CVECTOR_FREE and MPIRT_GROUP_FREE return MPIRT_CSET_NULL, MPIRT_CVECTOR_NULL and MPIRT_GROUP_NULL, respectively.

The following function creates an empty set.

MPIRT_CSET_CREATE (name, set)

IN	name	name of the container (MPIRT_STRING_NAME)
OUT	set	set container (ref. to MPIRT_CSET)

```
int MPIRT_Cset_create (MPIRT_String_name name, MPIRT_Cset *set)
```

```
MPIRT::Cset (const MPIRT::String_Name name)
```

The following function creates a vector of size size. This vector will have an empty entry for each size element of the container. An empty entry in an MPIRT_CVECTOR contains the value MPIRT_OBJECT_NULL.

MPIRT_CVECTOR_CREATE (name, size, vector)

IN	name	name of the container (MPIRT_STRING_NAME)
IN	size	initial size of the container (integer)
OUT	vector	vector container (ref. to MPIRT_CVECTOR)

```
int MPIRT_Cvector_create (MPIRT_String_name name, int size, MPIRT_Cvector *vector)
```

```
MPIRT::Cvector (const MPIRT::String_Name name, int size)
```

2.4.2 Generic Container Operation

The following operation returns the number of non-empty (not MPIRT_OBJECT_NULL) elements in a container. Recall that a set container contains only non-empty elements.

MPIRT_CONTAINER_ACCESS_COUNT (container, count)

IN	container	container (MPIRT_CONTAINER)
OUT	count	number of elements in the container (ref. to integer)

```
int MPIRT_Container_access_count (MPIRT_Container container, int *count)
```

```
int MPIRT::Container::Access_Count (int &count) const
```

⊤ (final)

2.4.3 Set Operations

⊥ (final)

The next operation adds an element to a set. If a reference to the element is in the set already, then the new reference to the same element is not added, an error MPIRT_ERR_CSET_DUPLICATE_MEMBER is raised, and the integrity of the container is not changed. ⊤ (final2)

An insertion of a NULL object of any type into a *S-container* has no effect on the container (no-op) and MPIRT_CSET_INSERT returns MPIRT_SUCCESS without any changes to the *S-container*.

⊥ (final2)

MPIRT_CSET_INSERT (set, element)

INOUT	set	set container (ref. to MPIRT_CSET)
IN	element	object to be inserted into the container (MPIRT_OBJECT)

```
int MPIRT_Cset_insert (MPIRT_Cset *set, MPIRT_Object element)
```

```
int MPIRT::Cset::Insert (const MPIRT::Object &element)
```

Advice to users. This can be a time-consuming operation. (*End of advice to users.*)

The next operation removes a reference to the object from a set and decreases the size of the set. The object that is to be removed from the set is passed into the MPIRT_CSET_REMOVE function. If the reference to the object is not in the set, then the error MPIRT_ERR_CSET_MEMBER_NOT_FOUND is returned, and the integrity of the container is unchanged. An attempt to remove of a NULL element of any type from a *S-container* returns an error ⊤ (final2) MPIRT_ERR_CSET_MEMBER_NOT_FOUND and does not change the integrity of the container. ⊥ (final2)

MPIRT_CSET_REMOVE (set, element)

INOUT	set	set container (ref. to MPIRT_CSET)
IN	element	object (ref. to MPIRT_OBJECT)

```
int MPIRT_Cset_remove (MPIRT_Cset *set, MPIRT_Object *element)
```

```
int MPIRT::Cset::Remove (const MPIRT::Object &element)
```

The next operation is used to iterate through the references to the elements in a set nondestructively. To retrieve the first element in the set, use the parameter MPIRT_OBJECT_NULL as the current element. When the last element of the set has been reached, MPIRT_OBJECT_NULL is returned in *nextelement*. Note that the order of retrieval of the elements may differ from one iteration through the set to the next. Also, if elements are inserted during an iteration pass, the newly-inserted elements may not be retrieved during this pass. If the reference to the current element is not in the set, then the error MPIRT_ERR_CSET_MEMBER_NOT_FOUND is returned, and the container's integrity is unchanged.

MPIRT_CSET_RETRIEVE_NEXT (set, currentelement, nextelement)

IN	set	set container (MPIRT_CSET)
IN	currentelement	current object (MPIRT_OBJECT)
OUT	nextelement	next object (ref. to MPIRT_OBJECT)

```
int MPIRT_Cset_retrieve_next (MPIRT_Cset set, MPIRT_Object currentelement,
             MPIRT_Object *nextelement)
```

```
int MPIRT::Cset::Retrieve_Next (const MPIRT::Object &currentelement,
             MPIRT::Object *&nextelement) const
```

The next operation queries a set container for a reference to an object.

MPIRT_CSET_QUERY (set, element, flag)

IN	set	set container (MPIRT_CSET)
IN	element	object (MPIRT_OBJECT)
OUT	flag	flag is MPIRT_TRUE if a reference to the object is in the set and MPIRT_FALSE otherwise (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Cset_query (MPIRT_Cset set, MPIRT_Object element, MPIRT_Boolean *flag)
```

```
int MPIRT::Cset::Query (const MPIRT::Object &element, MPIRT::Boolean &flag) const
```

⊤ (final)

2.4.4 Vector Base Operations

⊥ (final)

The following operation returns the size of a vector container regardless of what elements it contains.

MPIRT_CVECTOR_BASE_GET_SIZE (vector, size)

IN	vector	base vector container (MPIRT_CVECTOR_BASE)
OUT	size	size of the container (ref. to integer)

```
int MPIRT_Cvector_base_get_size (MPIRT_Cvector_base vector, int *size)
```

```
int MPIRT::Cvector_Base::Get_Size (int &size) const
```

The following function will resize an MPIRT_CVECTOR_BASE. If the new size is smaller than the old size, the higher-numbered indices will be truncated. If the new size is larger than the old size, the vector is grown, and the newly-added, higher-numbered entries are assigned the value MPIRT_OBJECT_NULL.

MPIRT_CVECTOR_BASE_SET_SIZE (vector, size)

INOUT	vector	base vector container (ref. to MPIRT_CVECTOR_BASE)
IN	size	size of the container (integer)

```
int MPIRT_Cvector_base_set_size (MPIRT_Cvector_base *vector, int size)
```

```
int MPIRT::Cvector_Base::Set_Size (int size)
```

The following operation returns the reference to the object at the specified index without removing it from the vector.

MPIRT_CVECTOR_BASE_RETRIEVE (vector, index, element)

IN	vector	base vector container (MPIRT_CVECTOR_BASE)
IN	index	index of the element to be retrieved from the container (integer)
OUT	element	MPI/RT object (ref. to MPIRT_OBJECT)

```
int MPIRT_Cvector_base_retrieve (MPIRT_Cvector_base vector, int index,
              MPIRT_Object *element)
```

```
int MPIRT::Cvector_Base::Retrieve (int index, MPIRT::Object *&element) const
```

The following operation returns the reference to the object at the specified index and replaces the entry with the value MPIRT_OBJECT_NULL.

MPIRT_CVECTOR_BASE_REMOVE (vector, index, element)

INOUT	vector	base vector container (ref. to MPIRT_CVECTOR_BASE)
IN	index	index of the element to be retrieved from the container (integer)
OUT	element	MPI/RT object (ref. to MPIRT_OBJECT)

```
int MPIRT_Cvector_base_remove (MPIRT_Cvector_base *vector, int index,
              MPIRT_Object *element)
```

```
int MPIRT::Cvector_Base::Remove (int index, MPIRT::Object *&element)
```

The following operation overwrites an element at the specified index in a vector.

MPIRT_CVECTOR_BASE_REPLACE (vector, index, element)

INOUT	vector	base vector container (ref. to MPIRT_CVECTOR_BASE)
IN	index	index of where the reference to the element is to be placed (integer)
IN	element	object to be placed into the vector container (MPIRT_OBJECT)

```
int MPIRT_Cvector_base_replace (MPIRT_Cvector_base vector, int index,
            MPIRT_Object *element)
```

```
int MPIRT::Cvector_Base::Replace (int index, const MPIRT::Object &element)
```

2.5 Groups

This section defines groups, the containers that define cohorts or collections of processes and provide integer-rank naming for such processes.

A **group** is a *V-container* of process identifiers[4]. These process identifiers are always opaque in MPI/RT, with programs referring to integer ranks of such identifiers within groups. Groups are not direct mechanisms for communication, but rather exist for organizing subsequent communication. They are integral in the definition of channels.

MPI/RT programs start in cohorts with a given "world" group (MPIRT_GROUP_WORLD). This world is the scope over which MPI/RT channel, event delivery abstraction objects and other committable objects are admitted collectively.

MPI/RT specifies correspondence between the rank in MPIRT_GROUP_WORLD and "task addresses," but task addresses are not used directly in MPI/RT, and task addresses themselves are abstractions of processes' (machine-dependent) most-primitive addresses. MPI/RT does not specify the function of "host processes" either. Other implementation-dependent, predefined groups may also be provided, such as those that refer to specialized I/O subsystems.

2.5.1 Group Definition

A **group** is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects, denoted in MPI/RT by MPIRT_GROUP, which is a *V-container*. There are no "holes"; ranks start from zero. The MPI/RT opaque entities contained in a group are called MPIRT_TASK_ADDRESS. Each process in a group is associated with an integer **rank**. Ranks are contiguous and start from zero. There is no explicit mechanism for serializing and unserializing groups.

There is a special pre-defined group, MPIRT_GROUP_EMPTY, which is a group with no members with the name "MPIRT_GROUP_EMPTY" set by implementation. The predefined constant MPIRT_GROUP_NULL is the value used for an invalid group.

The crucial group MPIRT_GROUP_WORLD is a global group that is defined after MPIRT_INIT successfully completes. This special group has the same rank assignments across all processes spawned by `mpirtrun`. The name parameter for the group MPIRT_GROUP_WORLD is set to "MPIRT_GROUP_WORLD" by an implementation. Each process also belongs to a special group

[4]Groups serve the same role as in MPI, but MPI/RT says more about their internal structure.

MPIRT_GROUP_SELF of cardinality one that contains only one process (itself). The name parameter for the group MPIRT_GROUP_SELF is set to "MPIRT_GROUP_SELF" by an implementation. ⊥ (final)

Advice to users. MPIRT_GROUP_EMPTY, which is a valid reference to an empty group, should not be confused with MPIRT_GROUP_NULL, which in turn is an invalid reference. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (*End of advice to users.*)

2.5.2 Group Management

There are no group manipulation operations defined in MPI/RT. Instead, users can use operations defined for MPIRT_CVECTOR_BASE to manipulate a group. Users can use a combination of the duplication operation and vector accessor operations to create new groups. All of these operations are local to a process. All processes in a group should create an identical group (vector base container with identical order of the MPIRT_TASK_ADDRESSes in it) before it is used. An implementation ensures that local references for a group among the group processes are properly matched during MPIRT_COMMIT.

MPIRT_GROUP_GET_RANK (group, rank)

IN	group	group (MPIRT_GROUP)
OUT	rank	rank of the calling process in the group, or MPIRT_RANK_UNDEFINED if the process is not a member (ref. to integer)

```
int MPIRT_Group_get_rank (MPIRT_Group group, int *rank)
```

```
int MPIRT::Group::Get_Rank (int &rank) const
```

⊤ (final2)

Rationale. While the standard does not define specific operations for creating groups the discussion below provides a strategy on how a user can create one via duplication of a group and using the MPIRT_CVECTOR_BASE manipulation operations.

Recall, that MPIRT_GROUP is a vector container that contains only non-empty elements of type MPIRT_TASK_ADDRESS (no holes) at any time. That means that for a group operations MPIRT_CVECTOR_BASE_GET_SIZE and MPIRT_CONTAINER_ACCESS_COUNT return the same value which is the size of the group. This also means that if an element is removed from the group using MPIRT_CVECTOR_BASE_REMOVE then the vector container is automatically decremented in size by one, and all elements of the group with the index higher than the index of the removed element are shifted to the "left" and their indices decremented by one. Since the group can not have any empty elements including MPIRT_OBJECT_NULL the attempt to increase the size of the group by using MPIRT_CVECTOR_BASE_SET_SIZE with the integer for size input parameter larger than the size of the group will not have any effect. That means that the operation will return success while the size of the group does not change. This should be viewed as a two step operation: the group vector enlarges the group container and fills new entries with MPIRT_OBJECT_NULLs and then when it is "converted" back to a group the empty elements are removed and the the group size is restored to the original one. Users can decrease the size of the group by using MPIRT_CVECTOR_BASE_SET_SIZE with the integer for size input parameter smaller than the size of the group. In this case all the group elements in the indices higher that the size parameter are removed and the group size shrinks to size.

Based on above statements users cannot create a group by adding new elements to an existing group, hence they can only remove or replace elements in the existing group but not add. Since all the processes have an identical local copy of the container for the MPIRT_GROUP_WOLRD it is a natural starting point for creating a new group for all interested processes.

First, a user duplicates the MPIRT_GROUP_WOLRD. Then the user removes all the ranks that are not a part of the new group from the copy. Users can either remove one rank at a time from its position using appropriate index for input parameter to MPIRT_CVECTOR_BASE_REMOVE, remembering that every removal decrements the index of all the higher ranks of the input group by one. Of course, a user can remove elements from the higher numbered rank to the lower numbered rank so they do not have to worry about index changes. Alternatively, they can move all the ranks that are part of the new group to the front and then resize the group to the size of the new group, thus removing all unneeded ranks from the back.

Recall, that both the elements and the order of these elements define the group. This means that if a user creates two groups with the same elements and preserving the relative order of the elements from the MPIRT_GROUP_WORLD then he or she just created two copies of the same group. In order to create a different group over the same process set every process must swap elements of the group the same way. This can be achieved by creating a variable of type MPIRT_TASK_ADDRESS for temporary storage and using MPIRT_CVECTOR_BASE_RETRIEVE and MPIRT_CVECTOR_BASE_REPLACE operations for a swap. (*End of rationale.*)

⊥ (final2)

⊤ (final) The duplication, free, equality testing operations and the get and set name parameter operations for event delivery objects are inherited from the general class of MPIRT_OBJECT (see Section 2.2).

⊥ (final)

2.6 Miscellaneous Objects

MPI/RT also defines other objects that have a body (system resources) but no operations defined on their bodies.

2.6.1 MPIRT_TASK_ADDRESS

MPI/RT defines one miscellaneous object that represents a computation unit (process) and that is ⊤ (final) automatically created by MPIRT_INIT and which forms the elements of the vector base container for MPIRT_GROUP. While no operation is defined to create an element of this object type, the general object operations: duplication (copy) (MPIRT_TASK_ADDRESS_DUP), free (MPIRT_TASK_ADDRESS_FREE), equality testing (MPIRT_TASK_ADDRESS_IS_EQUAL), and the get and set name parameter operations (MPIRT_TASK_ADDRESS_GET_NAME and MPIRT_TASK_ADDRESS_SET_NAME) are inherited by this object. ⊥ (final)

This is the opaque representation of a unit of computation's name in MPI/RT and is the con- ⊤ (final) stituent from which groups are assembled. The objects of type MPIRT_TASK_ADDRESS may have meaning only locally for a process and they do not necessarily have the same meaning (or value) for other processes. There is no user constructor for this opaque type. The name parameter is set to "TASK_ADDRESS_{number}" by an implementation when objects of this type are created by MPIRT_INIT, where {*number*} is the index of the MPIRT_TASK_ADDRESS object in the group (vector base container) MPIRT_GROUP_WORLD. ⊥ (final)

2.7 Summary

This section summarizes enumerations and type declarations that are introduced in this chapter.

The following depicts the implementation-independent aspects of the MPIRT_OBJECT:

```
namespace MPIRT
{
   class Object // key MPI/RT base class [pure virtual]
   {
   protected:
      Object();
      Object(const Object &rhs);
      Object &operator=(const Object &rhs);

public:     // this summarizes the interface [follows ICF]

      virtual  Object *Create_default(void)=0; // same as deflt constructor
      virtual  Object *Dup(void)=0;   // dup operator

      void Get_object_type(Object_type &type) const;
      void Get_name(String_Name &name) const;
      void Set_name(const String_Name &name);

      virtual Object_is_equal(const Object &comparator,
                              is_equal &type) const;
      virtual int operator==(const Object &comparator)
                              const; // Identical case
      virtual int operator!=(const Object &comparator)
                              const; // Not identical

      // Object decoration
      void Attach_attr(const Keyval &keyval, const Address &attribute_val);
      void Retrieve_attr(const Keyval &keyval, Address &attribute_val,
                         Boolean *flag) const;
      void Delete_attr(const Keyval &keyval);

      virtual  Free(void);   // equivalent to destructor
      virtual ~Object(void); // destructor

   protected:  // these parts are implementation-dependent
   private:
   };

   // other parts of namespace... */
```

```
};
```

The following depicts, for the subclasses (generically denoted *name-of-class*), the canonical form of these classes (and these inherit the public interface from the base class defined just above):

```
namespace MPIRT
{
   class name_of_class : public Object
   {
   public:
      // non-inherited interface:
      name_of_class(); // default constructor
      name_of_class(/* class-dependent args */);
      name_of_class(const name_of_class &rhs);
      name_of_class &operator=(const name_of_class &rhs);

      // inherited interface:
      virtual Object *Create_default(void);

   protected:  // these parts are implementation-dependent
   private:
   };
};
```

⊤ (final2)

⊥ (final2)

Miscellaneous Definitions		
Item	*What is it*	*Defined as:*
MPIRT_OBJECT	base object for all MPI/RT objects	object
MPIRT_OBJECT_NULL	NULL object	NULL object
MPIRT_STRING_NAME_NULL	NULL String_name	NULL string
MPIRT_KEYVAL	keyval object	keyval object
MPIRT_KEYVAL_NULL	NULL keyval object	NULL keyval object
MPIRT_KEYVAL_DUP_FN	function called when keyval is duped	typedef for a user function
MPIRT_KEYVAL_DELETE_FN	function called when keyval is deleted	typedef for a user function
MPIRT_KEYVAL_DUP_DEFLT_FN	default keyval dup function	enumeration
MPIRT_KEYVAL_DELETE_FN_EMPTY	default keyval delete function	enumeration
MPIRT_KEYVAL_DUP_FN_SHALLOW	predefined shallow keyval dup function	enumeration
MPIRT_KEYVAL_INVALID	invalid keyval	enumeration
MPIRT_CONTAINER_NULL	NULL container	NULL container
MPIRT_CSET	set object	set object
MPIRT_CSET_NULL	NULL set object	NULL set object
MPIRT_CVECTOR_BASE	base abstract vector object	base abstract vector object
MPIRT_CVECTOR	vector object	vector object
MPIRT_CVECTOR_NULL	NULL vector object	NULL vector object
MPIRT_GROUP	group object	group object
MPIRT_GROUP_WORLD	group object with all the processes	group object
MPIRT_GROUP_SELF	predefined group object with a single process	group object
MPIRT_GROUP_NULL	NULL group object	NULL group object
MPIRT_GROUP_EMPTY	empty group object	empty group object
MPIRT_TASK_ADDRESS	task address object	task address object
MPIRT_TASK_ADDRESS_NULL	NULL task address object	NULL task address object

Miscellaneous Bindings		
Item	*C Binding*	*C++ Binding*
MPIRT_OBJECT	MPIRT_Object	MPIRT::Object
MPIRT_OBJECT_NULL	MPIRT_OBJECT_NULL	MPIRT::OBJECT_NULL
MPIRT_STRING_NAME	MPIRT_String_name	MPIRT::String_Name
MPIRT_STRING_NAME_NULL	MPIRT_STRING_NAME_NULL	MPIRT::STRING_NAME_NULL
MPIRT_KEYVAL	MPIRT_Keyval	MPIRT::Keyval
MPIRT_KEYVAL_NULL	MPIRT_KEYVAL_NULL	MPIRT::KEYVAL_NULL
MPIRT_KEYVAL_DUP_FN	MPIRT_Keyval_dup_fn	MPIRT::Keyval::Dup_Fn
MPIRT_KEYVAL_DELETE_FN	MPIRT_Keyval_delete_fn	MPIRT::Keyval::Delete_Fn
MPIRT_KEYVAL_DUP_DEFLT_FN	MPIRT_KEYVAL_DUP_DEFLT_FN	MPIRT::Keyval::DUP_DEFLT_FN
MPIRT_KEYVAL_DELETE_FN_EMPTY	MPIRT_KEYVAL_DELETE_FN_EMPTY	MPIRT::Keyval::DELETE_FN_EMPTY
MPIRT_KEYVAL_DUP_FN_SHALLOW	MPIRT_KEYVAL_DUP_FN_SHALLOW	MPIRT::Keyval::DUP_FN_SHALLOW
MPIRT_KEYVAL_INVALID	MPIRT_KEYVAL_INVALID	MPIRT::KEYVAL_INVALID
MPIRT_CONTAINER_NULL	MPIRT_CONTAINER_NULL	MPIRT::CONTAINER_NULL
MPIRT_CSET	MPIRT_Cset	MPIRT::Cset
MPIRT_CSET_NULL	MPIRT_CSET_NULL	MPIRT::CSET_NULL
MPIRT_CVECTOR_BASE	MPIRT_Cvector_base	MPIRT::Cvector_Base
MPIRT_CVECTOR	MPIRT_Cvector	MPIRT::Cvector
MPIRT_CVECTOR_NULL	MPIRT_CVECTOR_NULL	MPIRT::CVECTOR_NULL
MPIRT_GROUP	MPIRT_Group	MPIRT::Group
MPIRT_GROUP_WORLD	MPIRT_GROUP_WORLD	MPIRT::GROUP_WORLD
MPIRT_GROUP_SELF	MPIRT_GROUP_SELF	MPIRT::GROUP_SELF
MPIRT_GROUP_NULL	MPIRT_GROUP_NULL	MPIRT::GROUP_NULL
MPIRT_GROUP_EMPTY	MPIRT_GROUP_EMPTY	MPIRT::GROUP_EMPTY
MPIRT_TASK_ADDRESS	MPIRT_Task_address	MPIRT::Task_Address
MPIRT_TASK_ADDRESS_NULL	MPIRT_TASK_ADDRESS_NULL	MPIRT::TASK_ADDRESS_NULL

Generic Object Operations		
Language Independent Specification	C	C++
MPIRT_OBJECT_GET_NAME	MPIRT_Object_get_name	MPIRT::Object::Get_Name
MPIRT_OBJECT_SET_NAME	MPIRT_Object_set_name	MPIRT::Object::Set_Name
MPIRT_OBJECT_GET_OBJECT_TYPE	MPIRT_Object_get_object_type	MPIRT::Object::Get_Type
MPIRT_OBJECT_DUP	MPIRT_Object_dup	MPIRT::Object MPIRT::Object::Dup MPIRT::Object::Operator=
MPIRT_OBJECT_FREE	MPIRT_Object_free	MPIRT::∼Object MPIRT::Object::Free
MPIRT_OBJECT_IS_EQUAL	MPIRT_Object_is_equal	MPIRT::Object::Is_equal
MPIRT_OBJECT_ATTACH_ATTR	MPIRT_Object_attach_attr	MPIRT::Object::Attach_Attr
MPIRT_OBJECT_RETRIEVE_ATTR	MPIRT_Object_retrieve_attr	MPIRT::Object::Retrieve_Attr
MPIRT_OBJECT_DELETE_ATTR	MPIRT_Object_delete_attr	MPIRT::Object::Delete_Attr

Keyval Operations		
Language Independent Specification	C	C++
MPIRT_KEYVAL_CREATE	MPIRT_Keyval_create	MPIRT::Keyval
MPIRT_KEYVAL_GET_DUP_FN	MPIRT_Keyval_get_dup_fn	MPIRT::Keyval::Get_Dup_Fn
MPIRT_KEYVAL_GET_DELETE_FN	MPIRT_Keyval_get_delete_fn	MPIRT::Keyval::Get_Delete_Fn
MPIRT_KEYVAL_GET_EXTRA_STATE	MPIRT_Keyval_get_extra_state	MPIRT::Keyval::Get_Extra_State
MPIRT_KEYVAL_SET_DUP_FN	MPIRT_Keyval_set_dup_fn	MPIRT::Keyval::Set_Dup_Fn
MPIRT_KEYVAL_SET_DELETE_FN	MPIRT_Keyval_set_delete_fn	MPIRT::Keyval::Set_Delete_Fn
MPIRT_KEYVAL_SET_EXTRA_STATE	MPIRT_Keyval_set_extra_state	MPIRT::Keyval::Set_Extra_State
MPIRT_KEYVAL_GET_NAME	MPIRT_Keyval_get_name	MPIRT::Keyval::Get_Name
MPIRT_KEYVAL_SET_NAME	MPIRT_Keyval_set_name	MPIRT::Keyval::Set_Name
MPIRT_KEYVAL_DUP	MPIRT_Keyval_dup	MPIRT::Keyval MPIRT::Keyval::Dup MPIRT::Keyval::Operator=
MPIRT_KEYVAL_FREE	MPIRT_Keyval_free	MPIRT::~Keyval MPIRT::Keyval::Free
MPIRT_KEYVAL_IS_EQUAL	MPIRT_Keyval_is_equal	MPIRT::Keyval::Is_equal
MPIRT_KEYVAL_ATTACH_ATTR	MPIRT_Keyval_attach_attr	MPIRT::Keyval::Attach_Attr
MPIRT_KEYVAL_RETRIEVE_ATTR	MPIRT_Keyval_retrieve_attr	MPIRT::Keyval::Retrieve_Attr
MPIRT_KEYVAL_DELETE_ATTR	MPIRT_Keyval_delete_attr	MPIRT::Keyval::Delete_Attr

Generic Container Operations		
Language Independent Specification	C	C++
MPIRT_CONTAINER_ACCESS_COUNT	MPIRT_Container_access_count	MPIRT::Container::Access_Count
MPIRT_CONTAINER_GET_NAME	MPIRT_Container_get_name	MPIRT::Container::Get_Name
MPIRT_CONTAINER_SET_NAME	MPIRT_Container_set_name	MPIRT::Container::Set_Name
MPIRT_CONTAINER_IS_EQUAL	MPIRT_Container_is_equal	MPIRT::Container::Is_equal
MPIRT_CONTAINER_ATTACH_ATTR	MPIRT_Container_attach_attr	MPIRT::Container::Attach_Attr
MPIRT_CONTAINER_RETRIEVE_ATTR	MPIRT_Container_retrieve_attr	MPIRT::Container::Retrieve_Attr
MPIRT_CONTAINER_DELETE_ATTR	MPIRT_Container_delete_attr	MPIRT::Container::Delete_Attr

CSet Operations		
Language Independent Specification	C	C++
MPIRT_CSET_CREATE	MPIRT_Cset_create	MPIRT::Cset
MPIRT_CSET_INSERT	MPIRT_Cset_insert	MPIRT::Cset::Insert
MPIRT_CSET_REMOVE	MPIRT_Cset_remove	MPIRT::Cset::Remove
MPIRT_CSET_RETRIEVE_NEXT	MPIRT_Cset_retrieve_next	MPIRT::Cset::Retrieve_Next
MPIRT_CSET_QUERY	MPIRT_Cset_query	MPIRT::Cset::Query
MPIRT_CSET_GET_NAME	MPIRT_Cset_get_name	MPIRT::Cset::Get_Name
MPIRT_CSET_SET_NAME	MPIRT_Cset_set_name	MPIRT::Cset::Set_Name
MPIRT_CSET_DUP	MPIRT_Cset_dup	MPIRT::Cset MPIRT::Cset::Dup MPIRT::Cset::Operator=
MPIRT_CSET_FREE	MPIRT_Cset_free	MPIRT::∼Cset MPIRT::Cset::Free
MPIRT_CSET_IS_EQUAL	MPIRT_Cset_is_equal	MPIRT::Cset::Is_equal
MPIRT_CSET_ATTACH_ATTR	MPIRT_Cset_attach_attr	MPIRT::Cset::Attach_Attr
MPIRT_CSET_RETRIEVE_ATTR	MPIRT_Cset_retrieve_attr	MPIRT::Cset::Retrieve_Attr
MPIRT_CSET_DELETE_ATTR	MPIRT_Cset_delete_attr	MPIRT::Cset::Delete_Attr

CVector Base Operations		
Language Independent Specification	C	C++
MPIRT_CVECTOR_BASE_GET_SIZE	MPIRT_Cvector_base_get_size	MPIRT::Cvector_Base::Get_Size
MPIRT_CVECTOR_BASE_SET_SIZE	MPIRT_Cvector_base_set_size	MPIRT::Cvector_Base::Set_Size
MPIRT_CVECTOR_BASE_RETRIEVE	MPIRT_Cvector_base_retrieve	MPIRT::Cvector_Base::Retrieve
MPIRT_CVECTOR_BASE_REMOVE	MPIRT_Cvector_base_remove	MPIRT::Cvector_Base::Remove
MPIRT_CVECTOR_BASE_REPLACE	MPIRT_Cvector_base_replace	MPIRT::Cvector_Base::Replace
MPIRT_CVECTOR_BASE_GET_NAME	MPIRT_Cvector_base_get_name	MPIRT::Cvector_Base::Get_Name
MPIRT_CVECTOR_BASE_SET_NAME	MPIRT_Cvector_base_set_name	MPIRT::Cvector_Base::Set_Name
MPIRT_CVECTOR_BASE_IS_EQUAL	MPIRT_Cvector_base_is_equal	MPIRT::Cvector_Base::Is_equal
MPIRT_CVECTOR_BASE_ATTACH_ATTR	MPIRT_Cvector_base_attach_attr	MPIRT::Cvector_Base::Attach_Attr
MPIRT_CVECTOR_BASE_RETRIEVE_ATTR	MPIRT_Cvector_base_retrieve_attr	MPIRT::Cvector_Base::Retrieve_Attr
MPIRT_CVECTOR_BASE_DELETE_ATTR	MPIRT_Cvector_base_delete_attr	MPIRT::Cvector_Base::Delete_Attr

CVector Operations		
Language Independent Specification	C	C++
MPIRT_CVECTOR_CREATE	MPIRT_Cvector_create	MPIRT::Cvector
MPIRT_CVECTOR_GET_SIZE	MPIRT_Cvector_get_size	MPIRT::Cvector::Get_Size
MPIRT_CVECTOR_SET_SIZE	MPIRT_Cvector_set_size	MPIRT::Cvector::Set_Size
MPIRT_CVECTOR_RETRIEVE	MPIRT_Cvector_retrieve	MPIRT::Cvector::Retrieve
MPIRT_CVECTOR_REMOVE	MPIRT_Cvector_remove	MPIRT::Cvector::Remove
MPIRT_CVECTOR_REPLACE	MPIRT_Cvector_replace	MPIRT::Cvector::Replace
MPIRT_CVECTOR_GET_NAME	MPIRT_Cvector_get_name	MPIRT::Cvector::Get_Name
MPIRT_CVECTOR_SET_NAME	MPIRT_Cvector_set_name	MPIRT::Cvector::Set_Name
MPIRT_CVECTOR_DUP	MPIRT_Cvector_dup	MPIRT::Cvector MPIRT::Cvector::Dup MPIRT::Cvector::Operator=
MPIRT_CVECTOR_FREE	MPIRT_Cvector_free	MPIRT::∼Cvector MPIRT::Cvector::Free
MPIRT_CVECTOR_IS_EQUAL	MPIRT_Cvector_is_equal	MPIRT::Cvector::Is_equal
MPIRT_CVECTOR_ATTACH_ATTR	MPIRT_Cvector_attach_attr	MPIRT::Cvector::Attach_Attr
MPIRT_CVECTOR_RETRIEVE_ATTR	MPIRT_Cvector_retrieve_attr	MPIRT::Cvector::Retrieve_Attr
MPIRT_CVECTOR_DELETE_ATTR	MPIRT_Cvector_delete_attr	MPIRT::Cvector::Delete_Attr

Group Operations		
Language Independent Specification	C	C++
MPIRT_GROUP_GET_SIZE	MPIRT_Group_get_name	MPIRT::Group::Get_Name
MPIRT_GROUP_SET_SIZE	MPIRT_Group_set_name	MPIRT::Group::Set_Name
MPIRT_GROUP_RETRIEVE	MPIRT_Group_retrieve	MPIRT::Group::Retrieve
MPIRT_GROUP_REMOVE	MPIRT_Group_remove	MPIRT::Group::Remove
MPIRT_GROUP_REPLACE	MPIRT_Group_replace	MPIRT::Group::Replace
MPIRT_GROUP_GET_NAME	MPIRT_Group_get_name	MPIRT::Group::Get_Name
MPIRT_GROUP_SET_NAME	MPIRT_Group_set_name	MPIRT::Group::Set_Name
MPIRT_GROUP_DUP	MPIRT_Group_dup	MPIRT::Group MPIRT::Group::Dup MPIRT::Group::Operator=
MPIRT_GROUP_FREE	MPIRT_Group_free	MPIRT::∼Group MPIRT::Group::Free
MPIRT_GROUP_IS_EQUAL	MPIRT_Group_is_equal	MPIRT::Group::Is_equal
MPIRT_GROUP_ATTACH_ATTR	MPIRT_Group_attach_attr	MPIRT::Group::Attach_Attr
MPIRT_GROUP_RETRIEVE_ATTR	MPIRT_Group_retrieve_attr	MPIRT::Group::Retrieve_Attr
MPIRT_GROUP_DELETE_ATTR	MPIRT_Group_delete_attr	MPIRT::Group::Delete_Attr

Task Address Operations		
Language Independent Specification	C	C++
MPIRT_TASK_ADDRESS_GET_NAME	MPIRT_Task_address_get_name	MPIRT::Task_Address::Get_Name
MPIRT_TASK_ADDRESS_SET_NAME	MPIRT_Task_address_set_name	MPIRT::Task_Address::Set_Name
MPIRT_TASK_ADDRESS_DUP	MPIRT_Task_address_dup	MPIRT::Task_Address MPIRT::Task_Address::Dup MPIRT::Task_Address::Operator=
MPIRT_TASK_ADDRESS_IS_EQUAL	MPIRT_Task_address_is_equal	MPIRT::Task_Address::Is_equal
MPIRT_TASK_ADDRESS_ATTACH_ATTR	MPIRT_Task_address_attach_attr	MPIRT::Task_Address::Attach_Attr
MPIRT_TASK_ADDRESS_RETRIEVE_ATTR	MPIRT_Task_address_retrieve_attr	MPIRT::Task_Address::Retrieve_Attr
MPIRT_TASK_ADDRESS_DELETE_ATTR	MPIRT_Task_address_delete_attr	MPIRT::Task_Address::Delete_Attr

Chapter 3

Dataspecs and Types

Current Status: First formal vote passed 11/0/0 on 6/98.
Second formal vote passed 9/0/3 on 7/98.
Third formal vote passed 7/0/0 on 12/3/98.

3.1 Dataspecs

MPI/RT provides a rich set of dataspecs that map to all basic data types in C and C++ as well as several that are MPI/RT specific (see Table 3.1). A dataspec provides type information of the data in buffers (see Section 5.2).

MPI/RT Dataspec	mapped to C type	mapped to C++ type
MPIRT_DATASPEC_CHAR	`signed char`	`signed char`
MPIRT_DATASPEC_SHORT	`signed short int`	`signed short int`
MPIRT_DATASPEC_INT	`signed int`	`signed int`
MPIRT_DATASPEC_LONG	`signed long int`	`signed long int`
MPIRT_DATASPEC_UNSIGNED_CHAR	`unsigned char`	`unsigned char`
MPIRT_DATASPEC_UNSIGNED_SHORT	`unsigned short int`	`unsigned short int`
MPIRT_DATASPEC_UNSIGNED	`unsigned int`	`unsigned int`
MPIRT_DATASPEC_UNSIGNED_LONG	`unsigned long int`	`unsigned long int`
MPIRT_DATASPEC_FLOAT	`float`	`float`
MPIRT_DATASPEC_DOUBLE	`double`	`double`
MPIRT_DATASPEC_LONG_DOUBLE	`long double`	`long double`
MPIRT_DATASPEC_BYTE		
MPIRT_DATASPEC_BOOLEAN	`MPIRT_Boolean`	`MPIRT::Boolean`
MPIRT_DATASPEC_TIME	`MPIRT_Time`	`MPIRT::Time`
MPIRT_DATASPEC_INT64	`MPIRT_Int64`	`MPIRT::Int64`

Table 3.1: MPI/RT dataspec mappings onto C and C++. Dataspecs are not type declarators, but rather type descriptors.

The above types are defined in terms of built-in types of C and C++ and, hence, have standard operations defined on them by the language (see [3, 7, 15]). ⊤ (final) ⊥ (final)

The type denoted by MPIRT_DATASPEC_BYTE consists of a byte (8 binary digits)[1]. A byte is uninterpreted and is, in general, distinct from a character. Different machines may have different representations for characters or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines.

3.1.1 Operations on Dataspec

There is only one operation defined on dataspecs that allows users to find the size of the dataspec in bytes.

MPIRT_DATASPEC_GET_SIZE (dataspec, size)

IN	dataspec	MPI/RT dataspec (MPIRT_DATASPEC)
OUT	size	dataspec size (ref. to integer)

```
int MPIRT_Dataspec_get_size (MPIRT_Dataspec dataspec, int *size)
```

```
int MPIRT::Dataspec::Get_Size (int &size) const
```

MPIRT_DATASPEC_GET_SIZE returns the total size, in bytes, of the minimal buffer required to store a unit of data as described by the `dataspec`. This is the total size of a message buffer that would be created for this dataspec.

3.2 Predefined MPI/RT Types

Any description of “enumeration” signifies that the value is one of some number of predefined values typically defined by equivalent methods such as C compiler directives of *enum* or *#define*.

3.2.1 MPIRT_BOOLEAN

The following boolean values are defined for MPI/RT:

- MPIRT_TRUE
- MPIRT_FALSE

These values can be used in normal, logical operations native to the programming language being used. In C they are mapped to integers, while in C++ they are mapped to type `bool`.

Advice to implementors. Here is a simple definition of boolean in C [7].

```
typedef int MPIRT_BOOLEAN;
#define MPIRT_TRUE (1)
#define MPIRT_FALSE (0)
```

(*End of advice to implementors.*)

[1] MPI/RT defines MPIRT_BYTE the same way as MPI-1 defines MPI_BYTE.

MPI/RT type	Description	C type	C++ type
MPIRT_BOOLEAN	logical	int	bool
MPIRT_STRING_NAME	Sequence of up to 127 non-null characters	char[128]	char[128]
MPIRT_INT64	64 bit integer	MPIRT_Int64 – either a 64-bit integer or two contiguous 32-bit integers	MPIRT::Int64 – either a 64-bit integer or two contiguous 32-bit integers
MPIRT_TIME	Holds time quantities in nanoseconds	MPIRT_Int64	MPIRT::Int64
MPIRT_TIME_SPEC	Structure with two fields: first field is enumeration with two values, MPIRT_ABSOLUTE and MPIRT_RELATIVE; second field is of type MPIRT_TIME	structure MPIRT_Time_spec	structure MPIRT::Time_Spec
MPIRT_ADDRESS	address (choice)	void *	void *
MPIRT_CONDITION	enumeration	int	int
MPIRT_BUFITER_POLICY	enumeration	int	int
MPIRT_SYNC_FLAG	enumeration	int	int
MPIRT_CHANNEL_DIRECTION	enumeration	int	int
MPIRT_CHANNEL_STATUS	enumeration	int	int
MPIRT_OBJECT_TYPE	enumeration	int	int or static class instantiation

Table 3.2: Predefined MPI/RT types

3.2.2 MPIRT_STRING_NAME

A string is a sequence of up to 127 characters that is null terminated. Its implementation in `C` follows this language convention, while in `C++`, the implementation is opaque, but commensurable easily with the `C` format. A string inherits all the operation from the language, including comparison, concatenation, and others.

3.2.3 MPIRT_INT64

MPI/RT defines the type MPIRT_INT64 to facilitate portable time functionality on systems that cannot natively handle 64-bit integer values. On these systems, this type is equivalent to two contiguous 32-bit integers. The basic arithmetic operations on the MPIRT_INT64 must also be supported to support these time values. On systems that can handle 64-bit integer values natively, this type can be a `typedef` to `long long int`, and these functions can be macros or inline functions.

The following two functions are used to assign a value to a variable of type MPIRT_INT64. Platforms that support 64-bit arithmetic must implement both versions of the assignment functions. All platforms must support the 32-bit assignment function.

⊤ (final2)

MPIRT_INT64_ASSIGN64 (value, variable)

IN	value	the value to be assigned to the MPIRT_INT64 variable (64-bit integer)
INOUT	variable	assigned variable (MPIRT_INT64)

`int MPIRT_Int64_assign64 (long long int value, MPIRT_Int64 *variable)`

`int MPIRT::Int64::Assign64 (long long int value, MPIRT::Int64 &variable)`

MPIRT_INT64_ASSIGN32 (upper_value, lower_value, variable)

IN	upper_value	the most significant 32-bits of the 64-bit value to be assigned to the MPIRT_INT64 variable (32-bit integer)
IN	lower_value	the least significant 32-bits of the 64-bit value to be assigned to the MPIRT_INT64 variable (32-bit unsigned integer)
INOUT	variable	assigned variable (MPIRT_INT64)

`int MPIRT_Int64_assign32 (int upper_value, unsigned int lower_value, MPIRT_Int64 *variable)`

`int MPIRT::Int64::Assign32 (int upper_value, unsigned int lower_value, MPIRT::Int64 &variable)`

MPIRT_INT64_ADD (addend, auggend, sum)

IN	addend	first number to be added (MPIRT_INT64)
IN	auggend	second number to be added (MPIRT_INT64)
OUT	sum	sum (MPIRT_INT64)

```
int MPIRT_Int64_add (MPIRT_Int64 addend, MPIRT_Int64 auggend, MPIRT_Int64 *sum)
```

```
int MPIRT::Int64::Add (MPIRT::Int64 addend, MPIRT::Int64 auggend, MPIRT::Int64
              &sum)
```

MPIRT_INT64_SUBTRACT (minuend, subtrahend, difference)

IN	minuend	the number from which the subtrahend is to be subtracted (MPIRT_INT64)
IN	subtrahend	the number that is to be subtracted from the minuend (MPIRT_INT64)
OUT	difference	the result of subtraction (MPIRT_INT64)

```
int MPIRT_Int64_subtract (MPIRT_Int64 minuend, MPIRT_Int64 subtrahend, MPIRT_Int64
              *difference)
```

```
int MPIRT::Int64::Subtract (MPIRT::Int64 minuend, MPIRT::Int64 subtrahend,
              MPIRT::Int64 &difference)
```

MPIRT_INT64_SCALAR_MULTIPLY (multiplicand, multiplier, product)

IN	multiplicand	the number that is to be multiplied by another (MPIRT_INT64)
IN	multiplier	the number by which the multiplicand is multiplied (integer)
OUT	product	the result of the scalar multiplication (MPIRT_INT64)

```
int MPIRT_Int64_scalar_multiply (MPIRT_Int64 multiplicand, int multiplier,
              MPIRT_Int64 *product)
```

```
int MPIRT::Int64::Scalar_Multiply (MPIRT::Int64 multiplicand, int multiplier,
              MPIRT::Int64 &product)
```

MPIRT_INT64_SCALAR_DIVIDE (dividend, divisor, quotient)

IN	dividend	the number to be divided (MPIRT_INT64)
IN	divisor	the scalar number by which the divisor is divided (integer)
OUT	quotient	the result of the division by the scalar (MPIRT_INT64)

```
int MPIRT_Int64_scalar_divide (MPIRT_Int64 dividend, int divisor, MPIRT_Int64
              *quotient)
```

```
int MPIRT::Int64::Scalar_Divide (const MPIRT::Int64 dividend, const int divisor,
              MPIRT::Int64 &quotient)
```

MPIRT_INT64_SCALAR_MODULO (dividend, divisor, remainder)

IN	dividend	the number to be divided (MPIRT_INT64)
IN	divisor	the scalar number by which the divisor is divided (integer)
OUT	remainder	remainder (MPIRT_INT64)

```
int MPIRT_Int64_scalar_modulo (MPIRT_Int64 dividend, int divisor, MPIRT_Int64
              *remainder)
```

```
int MPIRT::Int64::Scalar_Modulo (MPIRT::Int64 dividend, int divisor, MPIRT::Int64
              &remainder)
```

⊥ (final2)

The predicate of the comparison operation can be one of three values: a positive value means comparand greater than comparator, a negative value means comparand less than comparator, and zero means comparand = comparator.

> *Advice to implementors.* This is equivalent to performing a subtraction and using the difference to signify the relative ordering. (*End of advice to implementors.*)

MPIRT_INT64_COMPARE (comparand, comparator, predicate)

IN	comparand	the left-hand side of the comparison (MPIRT_INT64)
IN	comparator	the right-hand side of the comparison (MPIRT_INT64)
OUT	predicate	the result of the comparison (integer)

```
int MPIRT_Int64_compare (MPIRT_Int64 comparand, MPIRT_Int64 comparator, int
              *predicate)
```

```
int MPIRT::Int64::Compare (MPIRT::Int64 comparand, MPIRT::Int64 comparator, int
              &predicate)
```

> *Rationale.* Systems that cannot handle 64-bit integer values natively may be required to implement the MPIRT_INT64 type as an array of multiple integers of smaller size. (*End of rationale.*)

3.2.4 MPIRT_TIME

The MPIRT_TIME is defined as the time since the MPI/RT epoch, measured in nanoseconds, and is of type MPIRT_INT64.

3.2.5 MPIRT_TIME_SPEC

An MPIRT_TIME_SPEC has two fields, MPIRT_TIME_SPEC_TYPE and MPIRT_TIME_SPEC_TIME. MPIRT_TIME_SPEC_TYPE must have one of two values, MPIRT_ABSOLUTE or MPIRT_RELATIVE. The second field of MPIRT_TIME_SPEC should be of type MPIRT_TIME. In either case (MPIRT_ABSOLUTE or MPIRT_RELATIVE), the MPIRT_TIME_SPEC_TIME refers to the global synchronized clock, but in the relative case, an actual constraint is to be derived at runtime by adding MPIRT_TIME_SPEC_TIME to the value returned by as current as possible a read of the global synchronized clock. Note that even in the MPIRT_ABSOLUTE form, an actual time cannot necessarily be constructed until the value of the MPIRT_TIME_SPEC at the time of the execution of the call is known. A special value MPIRT_TIME_SPEC_BEST_EFFORT is defined for MPIRT_TIME_SPEC that is used to convey to an implementation that both fields of an MPIRT_TIME_SPEC are not significant.

The MPIRT_TIME_SPEC structure must have the two defined fields named as follows and be of the following format:

```
/* For C */

typedef enum
{
     MPIRT_ABSOLUTE,
     MPIRT_RELATIVE
} MPIRT_Time_spec_type;

typedef struct
{
     MPIRT_Time_spec_type  MPIRT_TIME_SPEC_TYPE;
     MPIRT_Time            MPIRT_TIME_SPEC_TIME;
     /* The implementation may include other, opaque fields */
} MPIRT_Time_spec;

// For C++

namespace MPIRT
{
    typedef enum
    {
        MPIRT_ABSOLUTE,
        MPIRT_RELATIVE
    } Time_spec_type;

    typedef struct
    {
        MPIRT::Time_spec_type  MPIRT_TIME_SPEC_TYPE;
        MPIRT::Time            MPIRT_TIME_SPEC_TIME;
        // The implementation may include other, opaque fields
        // and non-virtual methods here
    } Time_spec;
}
```

The MPIRT_TIME_SPEC has a special value of constant MPIRT_TIME_SPEC_BEST_EFFORT that represents the fact that no time requirement is requested by the application, but the implementation should perform the requested operation as soon as possible subject to guarantees for other quality-of-service requirements. The other special value for MPIRT_TIME_SPEC is the constant MPIRT_TIME_SPEC_IGNORE that represents that a parameter that uses this type can be ignored by an implementation (e.g. timeout, deadline, phase).

3.2.6 MPIRT_ADDRESS

The value of the type MPIRT_ADDRESS is a valid memory address that can be used to store message data. For both `C` and `C++`, MPIRT_ADDRESS is of language type `(void *)`.

3.3 Summary

This section summarizes enumerations and type declarations that are introduced in this chapter.

MPI/RT **Dataspecs**		
MPI/RT *Dataspec*	mapped to `C` type	mapped to `C++` type
MPIRT_DATASPEC_CHAR	`signed char`	`signed char`
MPIRT_DATASPEC_SHORT	`signed short int`	`signed short int`
MPIRT_DATASPEC_INT	`signed int`	`signed int`
MPIRT_DATASPEC_LONG	`signed long int`	`signed long int`
MPIRT_DATASPEC_UNSIGNED_CHAR	`unsigned char`	`unsigned char`
MPIRT_DATASPEC_UNSIGNED_SHORT	`unsigned short int`	`unsigned short int`
MPIRT_DATASPEC_UNSIGNED	`unsigned int`	`unsigned int`
MPIRT_DATASPEC_UNSIGNED_LONG	`unsigned long int`	`unsigned long int`
MPIRT_DATASPEC_FLOAT	`float`	`float`
MPIRT_DATASPEC_DOUBLE	`double`	`double`
MPIRT_DATASPEC_LONG_DOUBLE	`long double`	`long double`
MPIRT_DATASPEC_BYTE		
MPIRT_DATASPEC_BOOLEAN	`MPIRT_Boolean`	`MPIRT::Boolean`
MPIRT_DATASPEC_TIME	`MPIRT_Time`	`MPIRT::Time`
MPIRT_DATASPEC_INT64	`MPIRT_Int64`	`MPIRT::Int64`

$\top$ (final2)
$\perp$ (final2)
$\top$ (final)
$\perp$ (final)

MPI/RT Miscellaneous Bindings		
MPI/RT *Dataspec*	*C Binding*	*C++ Binding*
MPIRT_DATASPEC_CHAR	MPIRT_DATASPEC_CHAR	MPIRT::DATASPEC_CHAR
MPIRT_DATASPEC_SHORT	MPIRT_DATASPEC_SHORT	MPIRT::DATASPEC_SHORT
MPIRT_DATASPEC_INT	MPIRT_DATASPEC_INT	MPIRT::DATASPEC_INT
MPIRT_DATASPEC_LONG	MPIRT_DATASPEC_LONG	MPIRT::DATASPEC_LONG
MPIRT_DATASPEC_UNSIGNED_CHAR	MPIRT_DATASPEC_UNSIGNED_CHAR	MPIRT::DATASPEC_UNSIGNED_CHAR
MPIRT_DATASPEC_UNSIGNED_SHORT	MPIRT_DATASPEC_UNSIGNED_SHORT	MPIRT::DATASPEC_UNSIGNED_SHORT
MPIRT_DATASPEC_UNSIGNED	MPIRT_DATASPEC_UNSIGNED	MPIRT::DATASPEC_UNSIGNED
MPIRT_DATASPEC_UNSIGNED_LONG	MPIRT_DATASPEC_UNSIGNED_LONG	MPIRT::DATASPEC_UNSIGNED_LONG
MPIRT_DATASPEC_FLOAT	MPIRT_DATASPEC_FLOAT	MPIRT::DATASPEC_FLOAT
MPIRT_DATASPEC_DOUBLE	MPIRT_DATASPEC_DOUBLE	MPIRT::DATASPEC_DOUBLE
MPIRT_DATASPEC_LONG_DOUBLE	MPIRT_DATASPEC_LONG_DOUBLE	MPIRT::DATASPEC_LONG_DOUBLE
MPIRT_DATASPEC_BYTE	MPIRT_DATASPEC_BYTE	MPIRT::DATASPEC_BYTE
MPIRT_DATASPEC_BOOLEAN	MPIRT_DATASPEC_BOOLEAN	MPIRT::DATASPEC_BOOLEAN
MPIRT_DATASPEC_TIME	MPIRT_DATASPEC_TIME	MPIRT::DATASPEC_TIME
MPIRT_DATASPEC_INT64	MPIRT_DATASPEC_INT64	MPIRT::DATASPEC_INT64
MPIRT_TRUE	MPIRT_TRUE	MPIRT::TRUE
MPIRT_FALSE	MPIRT_FALSE	MPIRT::FALSE
MPIRT_DATASPEC_NULL	MPIRT_DATASPEC_NULL	MPIRT::DATASPEC_NULL
MPIRT_ABSOLUTE	MPIRT_ABSOLUTE	MPIRT::ABSOLUTE
MPIRT_RELATIVE	MPIRT_RELATIVE	MPIRT::RELATIVE
MPIRT_TIME_SPEC_IGNORE	MPIRT_TIME_SPEC_IGNORE	MPIRT::TIME_SPEC_IGNORE
MPIRT_TIME_SPEC_BEST_EFFORT	MPIRT_TIME_SPEC_BEST_EFFORT	MPIRT::TIME_SPEC_BEST_EFFORT
MPIRT_OBJECT_TYPE_PTCHANNEL	MPIRT_OBJECT_TYPE_PTCHANNEL	MPIRT::OBJECT_TYPE_PTCHANNEL
MPIRT_OBJECT_TYPE_BCAST_CHANNEL	MPIRT_OBJECT_TYPE_BCAST_CHANNEL	MPIRT::OBJECT_TYPE_BCAST_CHANNEL
MPIRT_OBJECT_TYPE_REDUCE_CHANNEL	MPIRT_OBJECT_TYPE_REDUCE_CHANNEL	MPIRT::OBJECT_TYPE_REDUCE_CHANNEL
MPIRT_OBJECT_TYPE_SCATTER_CHANNEL	MPIRT_OBJECT_TYPE_SCATTER_CHANNEL	MPIRT::OBJECT_TYPE_SCATTER_CHANNEL
MPIRT_OBJECT_TYPE_GATHER_CHANNEL	MPIRT_OBJECT_TYPE_GATHER_CHANNEL	MPIRT::OBJECT_TYPE_GATHER_CHANNEL
MPIRT_OBJECT_TYPE_BARRIER_CHANNEL	MPIRT_OBJECT_TYPE_BARRIER_CHANNEL	MPIRT::OBJECT_TYPE_BARRIER_CHANNEL
MPIRT_OBJECT_TYPE_BUFITER	MPIRT_OBJECT_TYPE_BUFITER	MPIRT::OBJECT_TYPE_BUFITER
MPIRT_OBJECT_TYPE_BUFFER	MPIRT_OBJECT_TYPE_BUFFER	MPIRT::OBJECT_TYPE_BUFFER

MPI/RT **Miscellaneous Bindings, Cont.**		
MPI/RT *Dataspec*	*C Binding*	*C++ Binding*
MPIRT_OBJECT_TYPE_HANDLER	MPIRT_OBJECT_TYPE_HANDLER	MPIRT::OBJECT_TYPE_HANDLER
MPIRT_OBJECT_TYPE_TRIGGER	MPIRT_OBJECT_TYPE_TRIGGER	MPIRT::OBJECT_TYPE_TRIGGER
MPIRT_OBJECT_TYPE_RECEPTOR	MPIRT_OBJECT_TYPE_RECEPTOR	MPIRT::OBJECT_TYPE_RECEPTOR
MPIRT_OBJECT_TYPE_CVECTOR	MPIRT_OBJECT_TYPE_CVECTOR	MPIRT::OBJECT_TYPE_CVECTOR
MPIRT_OBJECT_TYPE_CSET	MPIRT_OBJECT_TYPE_CSET	MPIRT::OBJECT_TYPE_CSET
MPIRT_OBJECT_TYPE_GROUP	MPIRT_OBJECT_TYPE_GROUP	MPIRT::OBJECT_TYPE_GROUP
MPIRT_OBJECT_TYPE_QOS_HANDLER	MPIRT_OBJECT_TYPE_QOS_HANDLER	MPIRT::OBJECT_TYPE_QOS_HANDLER
MPIRT_OBJECT_TYPE_QOS_RECEPTOR	MPIRT_OBJECT_TYPE_QOS_RECEPTOR	MPIRT::OBJECT_TYPE_QOS_RECEPTOR
MPIRT_OBJECT_TYPE_QOS_TRIGGER	MPIRT_OBJECT_TYPE_QOS_TRIGGER	MPIRT::OBJECT_TYPE_QOS_TRIGGER
MPIRT_OBJECT_TYPE_QOS_CHANNEL_TIME	MPIRT_OBJECT_TYPE_QOS_CHANNEL_TIME	MPIRT::OBJECT_TYPE_QOS_CHANNEL_TIME
MPIRT_OBJECT_TYPE_QOS_CHANNEL_PREV	MPIRT_OBJECT_TYPE_QOS_CHANNEL_PREV	MPIRT::OBJECT_TYPE_QOS_CHANNEL_PREV
MPIRT_OBJECT_TYPE_QOS_CHANNEL_PRTIEV	MPIRT_OBJECT_TYPE_QOS_CHANNEL_PRTIEV	MPIRT::OBJECT_TYPE_QOS_CHANNEL_PRTIEV
MPIRT_OBJECT_TYPE_TASK_ADDRESS	MPIRT_OBJECT_TYPE_TASK_ADDRESS	MPIRT::OBJECT_TYPE_TASK_ADDRESS
MPIRT_OBJECT_TYPE_DATASPEC	MPIRT_OBJECT_TYPE_DATASPEC	MPIRT::OBJECT_TYPE_DATASPEC
MPIRT_OBJECT_TYPE_KEYVAL	MPIRT_OBJECT_TYPE_KEYVAL	MPIRT::OBJECT_TYPE_KEYVAL
MPIRT_OBJECT_TYPE_PROBE	MPIRT_OBJECT_TYPE_PROBE	MPIRT::OBJECT_TYPE_PROBE
MPIRT_OBJECT_TYPE_REGISTRY	MPIRT_OBJECT_TYPE_REGISTRY	MPIRT::OBJECT_TYPE_REGISTRY
MPIRT_INT64	MPIRT_Int64	MPIRT::Int64
MPIRT_TIME	MPIRT_TIME	MPIRT::Time
MPIRT_TIME_NULL	MPIRT_TIME_NULL	MPIRT::TIME_NULL
MPIRT_TIME_IGNORE	MPIRT_TIME_IGNORE	MPIRT::TIME_IGNORE
MPIRT_TIME_SPEC	MPIRT_Time_spec	MPIRT::Time_Spec
MPIRT_DATASPEC	MPIRT_Dataspec	MPIRT::Dataspec
MPIRT_BOOLEAN	MPIRT_Boolean	MPIRT::Boolean
MPIRT_STRING_NAME	MPIRT_String_name	MPIRT::String_Name
MPIRT_ADDRESS	MPIRT_Address	MPIRT::Address
MPIRT_CONDITION	MPIRT_Condition	MPIRT::Condition
MPIRT_BUFITER_POLICY	MPIRT_Bufiter_policy	MPIRT::Bufiter::Policy
MPIRT_SYNC_FLAG	MPIRT_Sync_flag	MPIRT::Sync_Flag
MPIRT_CHANNEL_DIRECTION	MPIRT_Channel_direction	MPIRT::Channel::Direction
MPIRT_CHANNEL_STATUS	MPIRT_Channel_status	MPIRT::Channel::Status
MPIRT_OBJECT_TYPE	MPIRT_Object_type	MPIRT::Object_Type

MPI/RT **Types**			
MPI/RT *type*	*Description*	C type	C++ type
MPIRT_DATASPEC	dataspec object	`MPIRT_Dataspec`	`MPIRT::Dataspec`
MPIRT_BOOLEAN	logical	`int`	`bool`
MPIRT_STRING_NAME	Sequence of up to 127 non-null characters	`char[128]`	`char[128]`
MPIRT_INT64	64 bit integer	either a 64-bit integer or two contiguous 32-bit integers	either a 64-bit integer or two contiguous 32-bit integers
MPIRT_TIME	Holds time quantities in nanoseconds	`MPIRT_Int64`	`MPIRT::Int64`
MPIRT_TIME_SPEC	Structure with two fields: first field is enumeration with two values, MPIRT_ABSOLUTE and MPIRT_RELATIVE; second field is of type MPIRT_TIME	structure `MPIRT_Time_spec`	structure `MPIRT::Time_Spec`
MPIRT_ADDRESS	address (choice)	`void *`	`void *`
MPIRT_CONDITION	enumeration	`int`	`int`
MPIRT_BUFITER_POLICY	enumeration	`int`	`int`
MPIRT_SYNC_FLAG	enumeration	`int`	`int`
MPIRT_CHANNEL_DIRECTION	enumeration	`int`	`int`
MPIRT_CHANNEL_STATUS	enumeration	`int`	`int`
MPIRT_OBJECT_TYPE	enumeration	`int`	`int` or static class instantiation

MPI/RT **Constants**			
Constant	MPI/RT *Type*	C *Value*	C++ *Value*
MPIRT_TRUE	MPIRT_BOOLEAN	int 1	int 1
MPIRT_FALSE	MPIRT_BOOLEAN	int 0	int 0
MPIRT_DATASPEC_NULL	MPIRT_DATASPEC	implementation	implementation
MPIRT_ABSOLUTE	MPIRT_TIME_SPEC	implementation	implementation
MPIRT_RELATIVE	MPIRT_TIME_SPEC	implementation	implementation
MPIRT_TIME_SPEC_IGNORE	MPIRT_TIME_SPEC	implementation	implementation
MPIRT_TIME_SPEC_BEST_EFFORT	MPIRT_TIME_SPEC	implementation	implementation
MPIRT_OBJECT_TYPE_PTCHANNEL	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_BCAST_CHANNEL	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_REDUCE_CHANNEL	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_SCATTER_CHANNEL	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_GATHER_CHANNEL	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_BARRIER_CHANNEL	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_BUFITER	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_BUFFER	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_HANDLER	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_TRIGGER	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_RECEPTOR	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_CVECTOR	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_CSET	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_GROUP	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_QOS_HANDLER	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_QOS_RECEPTOR	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_QOS_TRIGGER	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_QOS_CHANNEL_TIME	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_QOS_CHANNEL_PREV	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_QOS_CHANNEL_PRTIEV	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_TASK_ADDRESS	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_DATASPEC	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_KEYVAL	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_PROBE	MPIRT_OBJECT_TYPE	implementation	implementation
MPIRT_OBJECT_TYPE_REGISTRY	MPIRT_OBJECT_TYPE	implementation	implementation

Operations		
Language Independent Specification	C	C++
MPIRT_DATASPEC_GET_SIZE	MPIRT_Dataspec_get_size	MPIRT::Dataspec::Get_Size
MPIRT_INT64_ASSIGN64	MPIRT_Int64_assign64	MPIRT::Int64::Assign64
MPIRT_INT64_ASSIGN32	MPIRT_Int64_assign32	MPIRT::Int64::Assign32
MPIRT_INT64_ADD	MPIRT_Int64_add	MPIRT::Int64::Add
MPIRT_INT64_SUBTRACT	MPIRT_Int64_subtract	MPIRT::Int64::Subtract
MPIRT_INT64_SCALAR_MULTIPLY	MPIRT_Int64_scalar_multiply	MPIRT::Int64::Scalar_Multiply
MPIRT_INT64_SCALAR_DIVIDE	MPIRT_Int64_scalar_divide	MPIRT::Int64::Scalar_Divide
MPIRT_INT64_SCALAR_MODULO	MPIRT_Int64_scalar_modulo	MPIRT::Int64::Scalar_Modulo
MPIRT_INT64_COMPARE	MPIRT_Int64_compare	MPIRT::Int64::Compare

Chapter 4

Event Delivery Abstraction and Handlers

Current Status: First vote on Section 4.2 passed by 7/1/0 on 5/20/98. First vote on Section 4.2 passed by 11/0/0 on 6/24/98. First vote on Section 4.3 passed by 9/0/1 on 6/24/98. Second vote on Section 4.3 passed by 8/0/3 on 7/22/98. Second vote on Section 4.2 passed by 7/0/5 on 7/23/98. Third vote passed 5/0/1 on 12/3/98.

4.1 Introduction

This chapter defines the elements of event delivery management that support asynchronous processing on local and remote processors based on the occurrence of conditions on objects across the MPI/RT concurrent program.

4.2 Event Delivery Abstraction

MPI/RT provides a mechanism that is able to initiate processing based on asynchronous event occurrences. Each (kind of) event is uniquely specified by a name and scope and may occur zero or more times during execution of the application. An event that occurs in one process can be processed in any other process(es). The event delivery abstraction has two parts: (event) *triggers* and (event) *receptors*.

For each process, users can specify a receptor with associated *handlers*. MPI/RT "connects" receptors and triggers based on event name and event scope. When an event occurs, a trigger is raised, MPI/RT delivers the event to all receptors for that event, and the receptor, in turn, schedules any handlers associated with it.

The triggers and receptors support the "receiver-based model" for control flow that allows users and implementors to schedule and control their activities by *guarding* them with receptors. The event delivery objects are committed along with channels, buffers, and other MPI/RT objects (see Chapter 12).

Users can attach handlers to the receptor for an event. These handlers can, in turn, start or stop any other user or MPI/RT activities.

An event name is scoped to a group and is unique within that scope. From the programmer's perspective, event triggers do not necessarily know if they are used in any receptors, and event receptors are not necessarily aware of where events are generated. MPI/RT provides the mechanism to match the two during the admission test. An *explicit receptor* is created by the programmer by using the MPIRT_RECEPTOR_CREATE function. An *implicit receptor* is created by the MPI/RT implementation on behalf of the user to support the requested quality-of-service (QoS), for example, to support channel QoS such as MPIRT_QOS_CHANNEL_PREV and MPIRT_QOS_CHANNEL_PRTIEV.

Similarly, triggers may be defined explicitly or implicitly. An *explicit trigger* is one created by the user by using the MPIRT_TRIGGER_CREATE function. Likewise, an *implicit trigger* is one created by the MPI/RT implementation on behalf of the user to support user notification of MPI/RT and system event occurrences such as the local completion of a channel message transfer.

In a single process, there can be at most one explicit receptor for a given event name. Implementations are allowed to create as many implicit receptors as they deem necessary within the bounds of the platform's resources. The MPI/RT specification does not define any bound on the number of explicit or implicit triggers that the user or the implementation may create; however, the implementation is allowed to fail and return an error if resources are exhausted. Figure 4.1 illustrates a possible relationship between triggers and receptors for the same event_name.

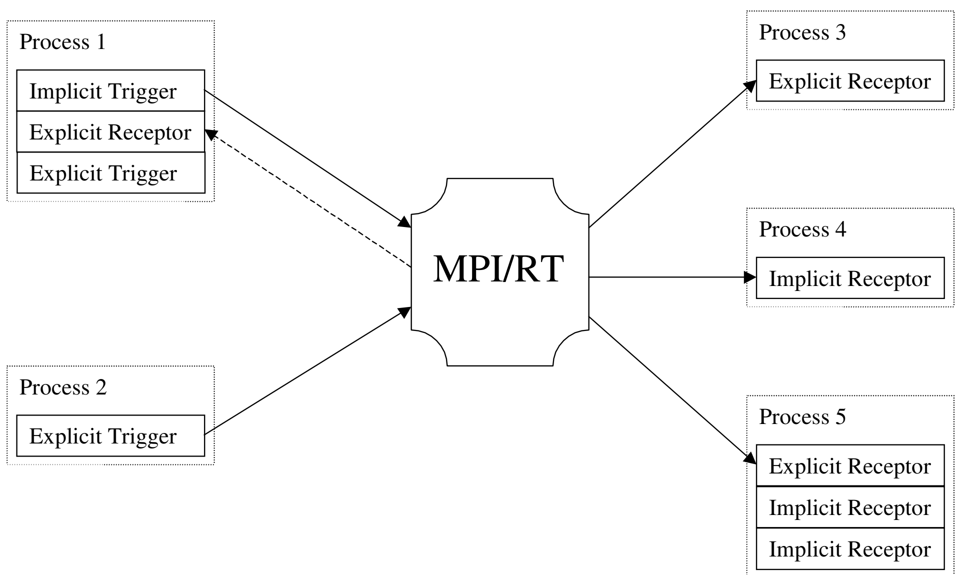

Figure 4.1: An example of possible arrangements of triggers and receptors.

Rationale. In order to preserve the order of execution of synchronous handlers with respect to each other, one explicit receptor at most is allowed per event name in a process. (*End of* *rationale.*)

Advice to users. If the user attempts to create an explicit receptor for an event name when one has already been created, the create function will return an error, MPIRT_RECEPTOR_NULL is returned for the receptor, and no new explicit receptor is created. The user must then call the query function MPIRT_RECEPTOR_RETRIEVE, using the event name to retrieve a reference to the existing explicit receptor. Once the user has this reference, additional handlers (both synchronous and asynchronous) can be added to that explicit receptor by using the receptor accessor functions and container manipulation functions. The standard QoS rules then apply at commit-time for handler execution ordering. Users can only add handlers to the explicit receptor when it is not committed. (*End of advice to users.*)

4.2.1 Event Naming

Event delivery objects have names that are significant to the semantics of both user and implementation behavior. These event names are used to attach receptors to specific triggers that are used to raise events in the program.

There is at most one implicit trigger (created by the implementation) over a given scope for each event name. The event name of implicit triggers includes the rank of the process where the object resides and the name of the object associated with the trigger. Users are allowed to create multiple (explicit) triggers for an event name within one or more processes within the given scope. Users are free to create triggers using the same name over the same scope as an implicit trigger. In that case, the name includes the process rank and the object name, but there is no requirement that this explicit trigger must reside in the same process or even the same node as the implicit trigger for that event. From the perspective of any given receptor for a particular event name, there is no distinction which trigger (either implicit or explicit) generated the event. The event name of an explicit trigger is not required to contain any information about the owning process. Note that explicit triggers generate conditions which can cause the creation of implicit triggers.

The object name is used to generate the event name(s) for the implicit trigger(s) based on that object name, the rank of the process that owns the object, and the condition name(s) of the event(s) on that object. For all non channel objects the rank is with respect to the group MPIRT_GROUP_WORLD. For the channel objects the rank is with respect to the group over which the channel is defined. Since the event name should be unique for creating a match between triggers and receptors, the object name must be unique with the process defined by the rank if there is a receptor for the event name. For all the objects this means that object name is unique with in the process. For channel objects that also means that the channel name is unique not only over the scope of the channel group but also over the scope of group MPIRT_GROUP_WORLD. This means that the channel name is unique with respect to all channels if there exist a receptor for the channel condition. For the object that do not have a condition defined and for objects that do not have a local or remote, implicit or explicit receptor defined for the object condition the object name is not required to be unique. The event name algorithm is as follows: {object name}:{rank of owning process}:{condition name}, where {object name} is the name of the object and {condition name} is defined in the appropriate object chapter. The name "radar_input_iterator:3:UNDERFLOW" is an example of an event name generated to identify the MPIRT_BUFITER_COND_UNDERFLOW condition generated by a buffer iterator with the object name "radar_input_iterator" on process rank 3 in the group MPIRT_GROUP_WORLD. A receptor associated with this event will use the event name "radar_input_iterator:3:UNDERFLOW" for matching.

An example of an explicit trigger with the event name "global_termination" which is located in any process would have "global_termination" as the event name. Any receptors associated with this event will use the event name "global_termination" for matching. For users who are interested in exactly which process within the group generated an event, the user can create explicit receptors, one for the event name of the implicit trigger for the raised condition of the explicit trigger on each process. In this case, the event name for the receptor will follow the rules for the implicit trigger. For example, "trigger_for_global_termination:4:RAISED" would be the event name used in the explicit receptor for the event occurring on the trigger named "trigger_for_global_termination" on process 4 of group MPIRT_GROUP_WORLD for the "global_termination" event. Note if process 4 raises its "global_termination" trigger, all receptors for event names "global_termination" *and* "trigger_for_global_termination:4:RAISED" will receive the event notification.

Rationale. Objects that can create implicit triggers must have group-unique names because the event names for the implicit triggers generated by those objects must be known to the user. (*End of rationale.*)

Advice to users. For channels, this event-naming scheme can be used to attach a receptor to any side (send or receive) of the channel by specifying the rank of the process that owns the desired channel endpoint since both endpoints of the channel must have the same object name.

A prefix can be used for object name to ensure uniqueness of the event name. For example, a layered library can use the library name as a prefix for all object names to avoid collisions for the event names with other libraries and user code. (*End of advice to users.*)

⊥ (final2)

Event Delivery

The issue of how the events are delivered from the event triggers to the event receptors that guard (constrain) an operation is left to the implementation.

Advice to implementors. The following methods are suggested for the event delivery:

1. Establish point-to-point channels between all the nodes of the group. MPI/RT will use these channels to deliver events. Using the registration (within the commit operation) together with the system and MPI/RT generated events, MPI/RT creates matching tables for all registered events. Upon receiving messages on these channels, MPI/RT checks whether the received event is sufficient to activate/deactivate an activity that is guarded by an implicit event receptor and schedules all handlers for the event receptor.

   Analogously, the implementation can establish a collective broadcast channel and broadcast events over it. Upon event arrival, the above schema can be used with respect to the receptors and handlers.
2. Use a method similar to the above but use signals (MPI/RT or system) for event delivery.
3. Use hardware-assisted event delivery.

(*End of advice to implementors.*)

4.2.2 Common Event Delivery Abstraction Operations

This section defines the common operations for triggers and receptors. The triggers and receptors share several attributes, namely event_name and event_scope. The common operations set and get these parameters for the event delivery objects.

Rationale. The event name allows users to create multiple triggers for the same event and allows multiple receptors for the same event. The scope for the event allows multiple libraries to coexist and allows users to scope the event delivery. (*End of rationale.*)

Both triggers and receptors can be either explicit or implicit. Explicit receptors can be matched to implicit triggers, and, likewise, implicit receptors can be matched to explicit triggers. The event-driven channel QoS specification defines implicit receptors to start or stop message transfers of a channel (see Chapter 11). Users can request an implicit receptor for an implicit trigger, thus creating a control flow for the entire application that does not require user explicit involvement.

The commit operation that registers event triggers and receptors is presented in Chapter 12.

The following operations allow users to query individual parameters of the event delivery object.

MPIRT_EVDEL_GET_EVENT_NAME (evdel, event_name)

IN	evdel	event delivery object (MPIRT_EVDEL)
OUT	event_name	name of the event that the trigger generates and the receptors receive (ref. to MPIRT_STRING_NAME)

```
int MPIRT_Evdel_get_event_name (MPIRT_Evdel evdel, MPIRT_String_name *event_name)
```

```
int MPIRT::Evdel::Get_Event_Name (MPIRT::String_Name &event_name) const
```

MPIRT_EVDEL_GET_SCOPE (evdel, scope)

IN	evdel	event delivery object (MPIRT_EVDEL)
OUT	scope	group that defines the scope of the event name (ref. to MPIRT_GROUP)

```
int MPIRT_Evdel_get_scope (MPIRT_Evdel evdel, MPIRT_Group *scope)
```

```
int MPIRT::Evdel::Get_Scope (MPIRT::Group *&scope) const
```

The next operations set individual parameters of an event delivery object.

MPIRT_EVDEL_SET_EVENT_NAME (evdel, event_name)

INOUT	evdel	event delivery object (ref. to MPIRT_EVDEL)
IN	event_name	name of the event that the trigger generates and the receptors receive (MPIRT_STRING_NAME)

```
int MPIRT_Evdel_set_event_name (MPIRT_Evdel *evdel, MPIRT_String_name event_name)
```

```
int MPIRT::Evdel::Set_Event_Name (const MPIRT::String_Name event_name)
```

MPIRT_EVDEL_SET_SCOPE (evdel, scope)

INOUT	evdel	event delivery object (ref. to MPIRT_EVDEL)
IN	scope	group that defines the scope of the event name (MPIRT_GROUP)

```
int MPIRT_Evdel_set_scope (MPIRT_Evdel *evdel, MPIRT_Group scope)
```

```
int MPIRT::Evdel::Set_Scope (const MPIRT::Group &scope)
```

The duplication, free, and equality testing operations and the get and set name parameter operations for event delivery objects are inherited from the general class of MPIRT_OBJECT (see Section 2.2).

There can be at most one explicit receptor object per event name (see section 4.2.4). Hence, the duplication operation cannot create another receptor with the same event name. To avoid this problem but still allow for a user to create a copy of a receptor, the duplication operation for receptor object automatically sets the event name for the copy to MPIRT_STRING_NAME_NULL.

4.2.3 Triggers

The following defines a trigger object of event delivery abstraction.

MPIRT_TRIGGER_CREATE (name, trigger_qos, event_name, scope, trigger)

IN	name	name of the trigger (MPIRT_STRING_NAME)
IN	trigger_qos	QoS for trigger or constant MPIRT_QOS_BEST_EFFORT (MPIRT_QOS_TRIGGER)
IN	event_name	name of the event that the trigger generates (MPIRT_STRING_NAME)
IN	scope	group that defines the scope of the event name (MPIRT_GROUP)
OUT	trigger	trigger (ref. to MPIRT_TRIGGER)

```
int MPIRT_Trigger_create (MPIRT_String_name name, MPIRT_Qos_trigger trigger_qos,
            MPIRT_String_name event_name, MPIRT_Group scope,
            MPIRT_Trigger *trigger)
```

```
MPIRT::Trigger (const MPIRT::String_Name name, const
            MPIRT::Qos_Trigger &trigger_qos, const MPIRT::String_Name event_name,
            const MPIRT::Group &scope)
```

Figure 4.2 defines the state transition diagram for the trigger object.

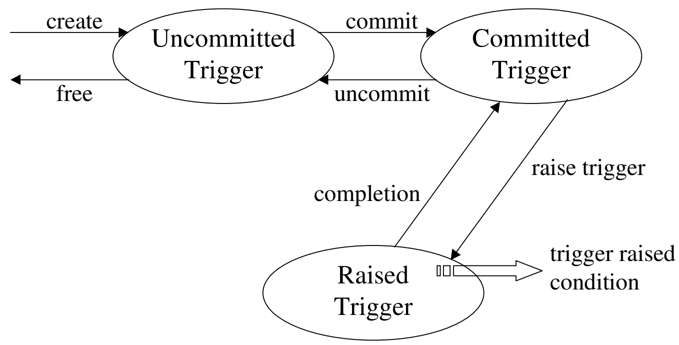

Figure 4.2: The state transition diagram for triggers.

Operations on Triggers

MPIRT_TRIGGER_GET_QOS (trigger, trigger_qos)

IN	trigger	trigger (MPIRT_TRIGGER)
OUT	trigger_qos	trigger's QoS (ref. to MPIRT_QOS_TRIGGER)

```
int MPIRT_Trigger_get_qos (MPIRT_Trigger trigger, MPIRT_Qos_Trigger *trigger_qos)
```

```
int MPIRT::Trigger::Get_Qos (MPIRT::Qos_Trigger *&trigger_qos) const
```

MPIRT_TRIGGER_SET_QOS (trigger, trigger_qos)

INOUT	trigger	trigger (ref. to MPIRT_TRIGGER)
IN	trigger_qos	trigger's QoS (MPIRT_QOS_TRIGGER)

```
int MPIRT_Trigger_set_qos (MPIRT_Trigger *trigger, MPIRT_Qos_Trigger trigger_qos)
```

```
int MPIRT::Trigger::Set_Qos (const MPIRT::Qos_Trigger &trigger_qos)
```

The following operation is provided for an application to raise an event occurrence for the trigger. This function is non-blocking. This operation can be used only for the committed trigger.

If the operation is called on a trigger which is not committed MPI/RT returns the MPIRT_ERR_OBJECT_NOT_COMMITTED error.

MPIRT_TRIGGER_RAISE (trigger)

INOUT	trigger	trigger (MPIRT_TRIGGER)

```
int MPIRT_Trigger_raise (MPIRT_Trigger *trigger)
```

```
int MPIRT::Trigger::Raise (void)
```

4.2.4 Event Receptors

⊥ (final2)

The MPIRT_RECEPTOR object can be masked to ignore incoming events. To support this functionality, the MPIRT_RECEPTOR has a state, enabled or disabled, for which the initial state can be specified in the create operation. Before the object is committed, this state controls the starting state after the commit. After a receptor is committed, these operations mask and unmask event delivery to the receptor. Three operations are defined for the MPIRT_RECEPTOR to query the state and to mask/unmask the receptor. These operations can be used for both committed and uncommitted receptors. One operation returns an explicit receptor from the system if one has been defined for a specific event name. This is not an operation on the receptor object itself but is a system function and is included here because of its relevance. Figure 4.3 defines the state transition diagram for the receptor object.

The following defines a receptor object of the event delivery abstraction.

MPIRT_RECEPTOR_CREATE (name, receptor_qos, event_name, scope, initial_state, handlers, receptor)

IN	name	name of the receptor (MPIRT_STRING_NAME)
IN	receptor_qos	QoS for the receptor (MPIRT_QOS_RECEPTOR), or constant object MPIRT_QOS_BEST_EFFORT (MPIRT_QOS_RECEPTOR)
IN	event_name	name of the event that raises the receptor (MPIRT_STRING_NAME)
IN	scope	group that defines the scope of the event name (MPIRT_GROUP)
IN	initial_state	the initial state of the receptor, MPIRT_TRUE is enabled and MPIRT_FALSE is disabled (MPIRT_BOOLEAN)
IN	handlers	a container of handler references to be invoked on the event arrival (MPIRT_CONTAINER)
OUT	receptor	receptor object for the event if one does not already exist or MPIRT_RECEPTOR_NULL if one already exists (ref. to MPIRT_RECEPTOR)

```
int MPIRT_Receptor_create (MPIRT_String_name name, MPIRT_Qos_receptor qos,
              MPIRT_String_name event_name, MPIRT_Group scope,
```

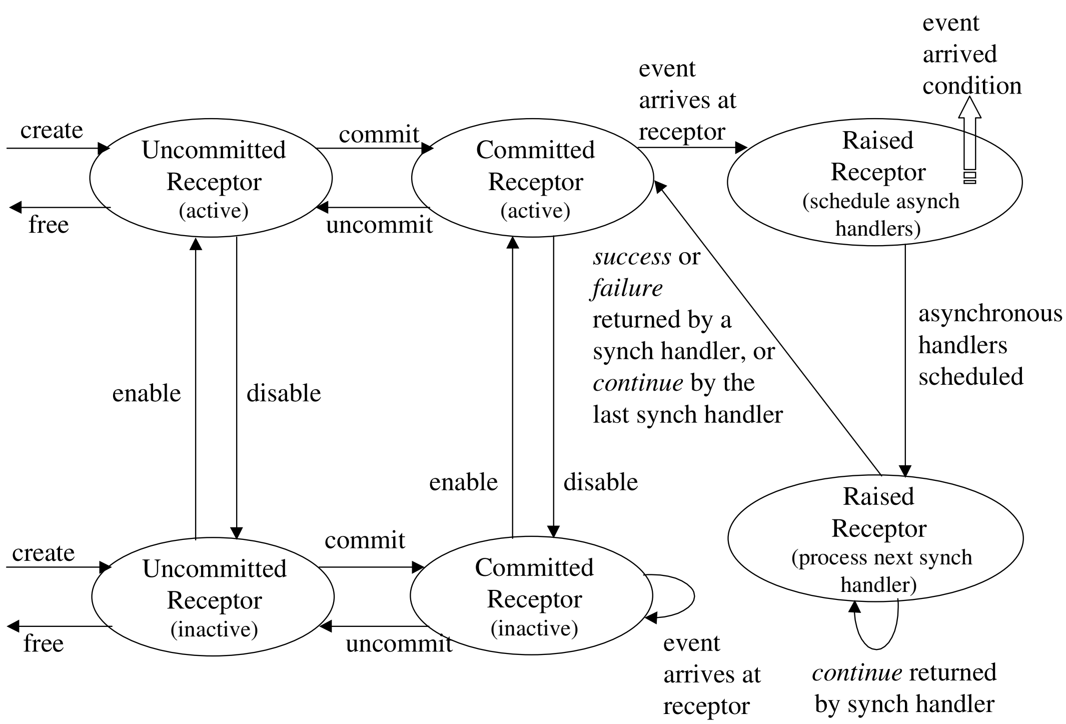

Figure 4.3: The state transition diagram for receptors.

```
                MPIRT_Boolean initial_state, MPIRT_Container handlers,
                MPIRT_receptor *receptor)

MPIRT::Receptor (const MPIRT::String_Name name, const
                MPIRT::Qos_Receptor &receptor_qos, const
                MPIRT::String_Name event_name, const MPIRT::Group &scope, const
                MPIRT::Boolean initial_state, const MPIRT::Container &handlers)
```

This function will return the error MPIRT_ERR_RECEPTOR_ALREADY_EXISTS in C (the analogous exception is thrown in C++), and the value of receptor will be MPIRT_RECEPTOR_NULL if a receptor for the event name already exists on this process. A receptor can not have an event name MPIRT_STRING_NAME_NULL at commit time if the receptor is in container of the objects to be committed.

If a process raises two events that are both received on another process, then the reception of the events in the other process is specified by the QoS of the receptors. If the two receptors have the same QoS, then the reception of them occurs in the order in which the events were raised on the raising process.

MPIRT_RECEPTOR_COND_EVENT_ARRIVED is the only condition which is defined for the MPIRT_RECEPTOR object.

Handlers associated with a receptor may be a mix of synchronous and asynchronous (see Section 4.3). Synchronous handlers are processed in the context of the receptor at the time an event is received. Asynchronous handlers are scheduled according to the QoS specified at the creation of each handler.

The order of processing of the handlers is controlled by the priority field of the QoS specified at handler creation. If handlers are given in a vector container, ties are broken in ascending order by the index of the handler in the vector container. If they are given in a set container, the order is not specified, and ties are processed in an arbitrary order.

Object conditions are classified as *recoverable* or *non-recoverable*. Recoverable conditions allow synchronous handlers to attempt to recover from the operation that raised the event trigger by returning a value to the implementation so that the operation can be retried. For conditions that are not recoverable, the implementation will not retry the operation that raised the event trigger. Only synchronous handlers that are local to the object that raised the event trigger may invoke a retry of the operation. Only operations defined by the standard can be recoverable and retryable. No user defined operation is recoverable . Asynchronous handlers, regardless of locality, and synchronous handlers that are non-local to the trigger-raising object cannot invoke a retry of the operation. Non-local synchronous handlers are processed as if the condition were non-recoverable. Return codes of non-local handlers of any type are not conveyed back to the operation that raised the event trigger.

The order of processing is as follows: Upon reception of an event, all asynchronous handlers are scheduled to be run. Once the asynchronous handlers have been scheduled, each synchronous handler is then called synchronously in the order determined above. When a synchronous handler function returns, its return code is examined to determine further actions by the implementation.

- If the value is MPIRT_CONTINUE, the next synchronous handler is called. If the value of MPIRT_CONTINUE is returned by the last synchronous handler of the receptor local to the object that triggered the event then the operation that raised the event trigger is not retried (even if it is recoverable) and the operation fails.
- If the value is MPIRT_SUCCESS, no further synchronous handlers are processed. In the case that the receptor is associated with a recoverable condition of the object that triggered the

event, and it is local to the receptor, the operation that raised the event trigger is retried. This allows recovery from this object condition. Recoverable conditions are retried only for the local receptor with synchronous handlers.

- If the value is MPIRT_FAILURE, no further processing of synchronous handlers occurs. The operation that raised the event trigger is not retried in this case, and the operation fails.

The return value for asynchronous handlers is not examined by the implementation for further action on recoverable conditions. An asynchronous handlers can only return the values of MPIRT_SUCCESS and MPIRT_FAILURE.

The return value of both synchronous and asynchronous handlers is used to raise the success and failure conditions associated with handlers (see Section 4.3).

Advice to users. This processing scheme can be used to achieve a variety of powerful instrumentation schemes and fault recovery on recoverable conditions. For example, users can reserve highest and lowest priorities of synchronous handlers for instrumentation in conjunction with light-weight handlers. Highest priorities can be used to instrument receptor activations. Lowest priorities can be used to instrument unrecoverable failures. Middle priorities can be used for normal processing and recovering from failures. (*End of advice to users.*)

MPIRT_RECEPTOR_GET_QOS (receptor, receptor_qos)

IN	receptor	receptor (MPIRT_RECEPTOR)
OUT	receptor_qos	receptor's QoS (ref. to MPIRT_QOS_RECEPTOR)

```
int MPIRT_Receptor_get_qos (MPIRT_Receptor receptor,
            MPIRT_Qos_receptor *receptor_qos)
```

```
int MPIRT::Receptor::Get_Qos (MPIRT::Qos_Receptor *&receptor_qos) const
```

MPIRT_RECEPTOR_GET_INITIAL_STATE (receptor, initial_state)

IN	receptor	receptor (MPIRT_RECEPTOR)
OUT	initial_state	MPIRT_TRUE for enabled, MPIRT_FALSE for disabled (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Receptor_get_initial_state (MPIRT_Receptor receptor,
            MPIRT_Boolean *initial_state)
```

```
int MPIRT::Receptor::Get_Initial_State (MPIRT::Boolean &initial_state) const
```

MPIRT_RECEPTOR_GET_HANDLERS (receptor, handlers)

IN	receptor	receptor (MPIRT_RECEPTOR)
OUT	handlers	a container of handler references to be invoked on the receptor conditions (ref. to MPIRT_CONTAINER)

```
int MPIRT_Receptor_get_handlers (MPIRT_Receptor receptor,
             MPIRT_Container *handlers)
```

```
int MPIRT::Receptor::Get_Handlers (MPIRT::Container *&handlers) const
```

MPIRT_RECEPTOR_SET_QOS (receptor, receptor_qos)

INOUT	receptor	receptor (ref. to MPIRT_RECEPTOR)
IN	receptor_qos	receptor's QoS (MPIRT_QOS_RECEPTOR)

```
int MPIRT_Receptor_set_qos (MPIRT_Receptor *receptor,
             MPIRT_Qos_receptor receptor_qos)
```

```
int MPIRT::Receptor::Set_Qos (const MPIRT::Qos_Receptor &receptor_qos)
```

MPIRT_RECEPTOR_SET_INITIAL_STATE (receptor, initial_state)

INOUT	receptor	receptor (MPIRT_RECEPTOR)
IN	initial_state	MPIRT_TRUE for enabled, MPIRT_FALSE for disabled (MPIRT_BOOLEAN)

```
int MPIRT_Receptor_set_initial_state (MPIRT_Receptor *receptor,
             MPIRT_Boolean initial_state)
```

```
int MPIRT::Receptor::Set_Initial_State (MPIRT::Boolean initial_state)
```

MPIRT_RECEPTOR_SET_HANDLERS (receptor, handlers)

INOUT	receptor	receptor (ref. to MPIRT_RECEPTOR)
IN	handlers	a container of handler references to be invoked on the receptor conditions (MPIRT_CONTAINER)

```
int MPIRT_Receptor_set_handlers (MPIRT_Receptor *receptor,
             MPIRT_Container handlers)
```

```
int MPIRT::Receptor::Set_Handlers (const MPIRT::Container &handlers)
```

⊥ (final2)

System Query Function for Receptor

The following function queries the system for a reference to an existing explicit receptor object if one exists. This function is an MPI/RT system call and not a member of the receptor object, but it is included here because of its relevance to the receptor functionality.

MPIRT_RETRIEVE_RECEPTOR (event_name, receptor)

IN	event_name	name of the event (MPIRT_STRING_NAME)
OUT	receptor	existing receptor (ref. to MPIRT_RECEPTOR)

```
int MPIRT_Retrieve_receptor (MPIRT_String_name event_name,
              MPIRT_Receptor *receptor)
```

```
int MPIRT::Retrieve_Receptor (MPIRT::String_Name event_name,
              MPIRT::Receptor *&receptor) const
```

If no explicit receptor exists for the event name in the process, the error MPIRT_ERR_NO_RECEPTOR_EXISTS is returned in C (respectively, the exception is thrown in C++), and the value of receptor returned is MPIRT_RECEPTOR_NULL.

Operations on Receptors

The following function queries the state of the receptor.

MPIRT_RECEPTOR_IS_ENABLED (receptor, flag)

IN	receptor	receptor (MPIRT_RECEPTOR)
OUT	flag	the state of the receptor, MPIRT_TRUE if the receptor is enabled or MPIRT_FALSE if the receptor is disabled (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Receptor_is_enabled (MPIRT_Receptor receptor, MPIRT_Boolean *flag)
```

```
int MPIRT::Receptor::Is_Enabled (MPIRT::Boolean &flag) const
```

The next two operations allow users to enable and disable the event receptor.

MPIRT_RECEPTOR_ENABLE (receptor, flag)

INOUT	receptor	receptor (ref. to MPIRT_RECEPTOR)
OUT	flag	previous state of the receptor, MPIRT_TRUE if previously enabled or MPIRT_FALSE if previously disabled (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Receptor_enable (MPIRT_Receptor *receptor, MPIRT_Boolean *flag)
```

```
int MPIRT::Receptor::Enable (MPIRT::Boolean &flag)
```

MPIRT_RECEPTOR_DISABLE (receptor, flag)

INOUT	receptor	receptor (ref. to MPIRT_RECEPTOR)
OUT	flag	previous state of the receptor, MPIRT_TRUE if previously enabled or MPIRT_FALSE if previously disabled (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Receptor_disable (MPIRT_Receptor *receptor, MPIRT_Boolean *flag)
```

```
int MPIRT::Receptor::Disable (MPIRT::Boolean &flag)
```

4.3 Handlers

Handlers are a useful mechanism for implementation of real-time applications. For example, they can be used for implementing error handling, instrumentation, scheduling of the event-driven paradigm, and, in conjunction with operating system timers, for the scheduling of the time-driven paradigm. Handlers are evaluated and committed in the admission test. Figure 4.4 defines the state transition diagram for the handler object.

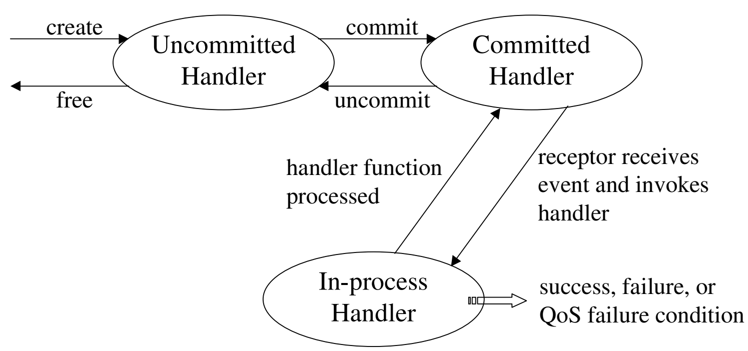

Figure 4.4: The state transition diagram for handlers.

4.3.1 Handler Constructors

The following section describes the handler object which is used in conjunction with receptors (see Section 4.2.4) for event processing.

MPIRT_HANDLER_CREATE (name, handler_fn, sync_flag, handler_qos, extra_state, handler)

IN	name	name of the handler (MPIRT_STRING_NAME)
IN	handler_fn	handler function to be invoked on the object condition (user-defined function MPIRT_HANDLER_FN)
IN	sync_flag	specifies synchronous (MPIRT_HANDLER_SYNC) or asynchronous (MPIRT_HANDLER_ASYNC) handler processing (MPIRT_SYNC_FLAG)
IN	handler_qos	QoS for the handler or constant MPIRT_QOS_BEST_EFFORT (MPIRT_QOS_HANDLER)
IN	extra_state	user-supplied parameters for the handler (MPIRT_ADDRESS)
OUT	handler	reference to the handler object (ref. to MPIRT_HANDLER)

```
int MPIRT_Handler_create (MPIRT_String_name name, MPIRT_Handler_fn *handler_fn,
              MPIRT_Sync_flag sync_flag, MPIRT_Qos_handler qos,
              MPIRT_Address extra_state, MPIRT_Handler *handler)
```

```
MPIRT::Handler (const MPIRT::String_Name name, MPIRT::Handler::Fn handler_fn,
              MPIRT::Sync_Flag sync_flag, const MPIRT::Qos_Handler &qos,
              MPIRT::Address extra_state)
```

A handler can be processed either *synchronously* or *asynchronously* with respect to the processing of the receptor to which it is attached. Synchronous handlers, created by specifying the value of the *sync_flag* as MPIRT_HANDLER_SYNC, are processed in the same context as the processing of the receptor – effectively as subroutine calls. Asynchronous handlers, created by using MPIRT_HANDLER_ASYNC for sync_flag, are processed according to their specified QoS after the processing of the receptor, possibly in a separate context from the receptor processing.

The processing of synchronous handlers is controlled by the value returned by the handler function, which must be one of the following three values: MPIRT_SUCCESS, MPIRT_FAILURE, or MPIRT_CONTINUE. Processing of synchronous vs. asynchronous handlers is discussed in detail in Section 4.2.4.

Handlers generate the following conditions for which the implementation creates implicit triggers. These are MPIRT_HANDLER_COND_SUCCESS, MPIRT_HANDLER_COND_FAILURE, and MPIRT_HANDLER_COND_QOS_FAILURE. Other handlers may be created and associated with receptors that are activated by one of these triggers. The names associated with these conditions are "SUCCESS", "FAILURE", and "QOS_ERROR", respectively, for event name generation. See Section 4.2.1 for the formula used to generate unique event names for use with receptors.

The correlation between the return value of the handler and the condition raised is defined as follows. If the handler returns:

MPIRT_SUCCESS then the condition MPIRT_HANDLER_COND_SUCCESS is raised for both synchronous and asynchronous handlers if there exists a committed receptor for that event name.

MPIRT_FAILURE then the condition MPIRT_HANDLER_COND_FAILURE is raised for both synchronous and asynchronous handlers if there exists a committed receptor for that event name.

MPIRT_CONTINUE then for the last synchronous handler the condition MPIRT_HANDLER_COND_FAILURE is raised if there exists a committed receptor for that event name. If it is not the last synchronous handler then no condition is raised.

The condition MPIRT_HANDLER_COND_QOS_ERROR is raised for both synchronous and asynchronous handlers when an implementation was not able to schedule the handler within the QoS specification requested by the user.

> *Rationale.* Recall that only synchronous handlers can return MPIRT_CONTINUE (see section 4.2.4). If all the local synchronous handlers for recoverable condition are not able to fix the problem which raised the event then the operation that caused the problem can not be recovered from. So, a user should be able to find out about it. Hence, if the return value of the last synchronous handler is MPIRT_CONTINUE then the condition MPIRT_HANDLER_COND_FAILURE is raised by the implementation. (*End of rationale.*)

The processing of synchronous and asynchronous handlers is determined by the QoS of the handler at commit time. The QoS of handlers is specified by two parameters: priority and timeout. For synchronous handlers, the priority must be MPIRT_PRIORITY_INHERIT to represent that synchronous handlers are running in the context of the receptor. For both synchronous and asynchronous handlers timeout should be specified by MPIRT_TIME_SPEC of the MPIRT_RELATIVE form (see Sections 3.2.5 and 11.3).

The following is a language-independent specification of what a user handler function definition should be. It is not a function defined in MPI/RT. The C and C++ definitions for this function are defined below.

MPIRT_HANDLER_FN (receptor, handler, obj, condition)

IN	receptor	receptor object (MPIRT_RECEPTOR)
IN	handler	handler object (MPIRT_HANDLER)
IN	obj	object for which the event has been triggered (MPIRT_OBJECT)
IN	condition	condition which triggered this event (integer)

```
typedef int (*MPIRT_Handler_fn) (MPIRT_Receptor receptor, MPIRT_Handler handler,
             MPIRT_Object obj, int condition);
```

```
typedef int (*MPIRT::Handler::Fn) (MPIRT::Receptor &receptor, MPIRT::Object &obj,
             int condition);
```

If the object that triggered the event is not local (instantiated in the same process) to the handler or if the triggering object is an explicit trigger, then the obj parameter will be MPIRT_OBJECT_NULL, and the condition parameter will be MPIRT_CONDITION_UNKNOWN.

> *Advice to users.* The extra_state that the user passes into the handlers can be retrieved in the handler_fn using the MPIRT_HANDLER_GET_EXTRA_STATE operation on the handler reference that is passed into the handler_fn. Any other parameters of the handlers can be also retrieved that way. (*End of advice to users.*)

4.3.2 Light-Weight Handler Functions

To create a light-weight handler, the creation function is used, and one of the predefined light-weight handler function values is passed in as the handler_fn argument. All light-weight handler functions return the value MPIRT_CONTINUE. The extra_state parameter is specified according to the desired light-weight operation, as defined below.

Basic Operations

The following six functions provide basic manipulations of an integer provided by the user. For each of these, the extra_state pointer is interpreted as a pointer to the signed integer to be manipulated.

SET_INT Sets the integer to the value 1.

CLEAR_INT Sets the integer to the value 0.

TOGGLE_INT Logical *not* operation on the integer.

COMPLEMENT_INT Bitwise *not* operation on the integer.

INCREMENT_INT Adds 1 to the integer.

DECREMENT_INT Subtracts 1 from the integer.

These are defined over C and C++ as follows.

```
/* C definitions */
MPIRT_HANDLER_FN_LW_SET_INT
MPIRT_HANDLER_FN_LW_CLEAR_INT
MPIRT_HANDLER_FN_LW_TOGGLE_INT
MPIRT_HANDLER_FN_LW_COMPLEMENT_INT
MPIRT_HANDLER_FN_LW_INCREMENT_INT
MPIRT_HANDLER_FN_LW_DECREMENT_INT

// C++ definitions
MPIRT::HANDLER::FN_LW_SET_INT
MPIRT::HANDLER::FN_LW_CLEAR_INT
MPIRT::HANDLER::FN_LW_TOGGLE_INT
MPIRT::HANDLER::FN_LW_COMPLEMENT_INT
MPIRT::HANDLER::FN_LW_INCREMENT_INT
MPIRT::HANDLER::FN_LW_DECREMENT_INT
```

For the next operation, the address of extra_state is interpreted as beginning with a structure of the form:

```
typedef struct mpirt_handler_fn_lw_mask_extra_info
{
   unsigned int target_int;
   unsigned int and_mask;
   unsigned int or_mask;
   unsigned int xor_mask;
} MPIRT_Handler_fn_lw_mask_extra_info;
```

The MPIRT_BITWISE_INT operation performs “and” followed by “or” followed by “xor” on the unsigned integer located at the pointer `target_int`.

```
/* C definition */
MPIRT_HANDLER_FN_LW_BITWISE_INT

// C++ definition
MPIRT::HANDLER::FN_LW_BITWISE_INT
```

⊤ (final2)

Advice to implementors. These light-weight operations can be implemented as enumerations, macros, or function pointers as long as they are resolved as being of type MPIRT_HANDLER_FN∗ so that compiler warnings do not occur.

They can be implemented as cases in a *switch* statement or by function calls, but high-quality implementations will implement them to be as light weight as possible. (*End of advice to implementors.*)

⊥ (final2)

Advice to users. Although the user may create a handler using a light-weight function and specify it to be synchronous or asynchronous, minimum overhead will be achieved when it is created as a synchronous handler. (*End of advice to users.*)

4.3.3 Handler Accessors

The following operations allow users to query individual parameters of the handler.

MPIRT_HANDLER_GET_HANDLER_FN (handler, handler_fn)

IN	handler	handler (MPIRT_HANDLER)
OUT	handler_fn	handler function to be invoked on the object condition (ref. to MPIRT_HANDLER_FN)

```
int MPIRT_Handler_get_handler_fn (MPIRT_Handler handler,
             MPIRT_Handler_fn *handler_fn)
```

```
int MPIRT::Handler::Get_Handler_Fn (MPIRT::Handler::Fn &handler_fn) const
```

The next operation returns the QoS information of the handler.

MPIRT_HANDLER_GET_QOS (handler, qos)

IN	handler	handler (MPIRT_HANDLER)
OUT	qos	QoS for the handler (ref. to MPIRT_QOS_HANDLER)

```
int MPIRT_Handler_get_qos (MPIRT_Handler handler, MPIRT_Qos_handler *qos)
```

⊤ (final2)

```
int MPIRT::Handler::Get_Qos (MPIRT::Qos_Handler *&qos) const
```

The following function returns the synchronization mode of the handler.

MPIRT_HANDLER_GET_SYNC (handler, sync)

IN	handler	handler (MPIRT_HANDLER)
OUT	sync	type of handler, either MPIRT_HANDLER_SYNC or MPIRT_HANDLER_ASYNC (enumeration)

```
int MPIRT_Handler_get_sync (MPIRT_Handler handler, MPIRT_Sync_flag *sync)
```

```
int MPIRT::Handler::Get_Sync(MPIRT::Sync_Flag &sync) const
```

The next operation returns the extra state information of the handler.

MPIRT_HANDLER_GET_EXTRA_STATE (handler, extra_state)

IN	handler	handler (MPIRT_HANDLER)
OUT	extra_state	the user-supplied parameters for the handler (ref. to MPIRT_ADDRESS)

```
int MPIRT_Handler_get_extra_state (MPIRT_Handler handler,
              MPIRT_Address *extra_state)
```

```
int MPIRT::Handler::Get_Extra_State (MPIRT::Address &extra_state) const
```

The following operations allow users to set individual parameters of the handler.

MPIRT_HANDLER_SET_HANDLER_FN (handler, handler_fn)

INOUT	handler	handler (ref. to MPIRT_HANDLER)
IN	handler_fn	handler function to be invoked on the object condition (MPIRT_HANDLER_FN)

```
int MPIRT_Handler_set_handler_fn (MPIRT_Handler *handler,
              MPIRT_Handler_fn handler_fn)
```

```
int MPIRT::Handler::Set_Handler_Fn (MPIRT::Handler::Fn handler_fn)
```

The following function sets the synchronization mode for the handler.

MPIRT_HANDLER_SET_SYNC (handler, sync)

INOUT	handler	handler (ref. to MPIRT_HANDLER)
IN	sync	type of handler, either MPIRT_HANDLER_SYNC or MPIRT_HANDLER_ASYNC (enumeration)

```
int MPIRT_Handler_set_sync (MPIRT_Handler *handler, MPIRT_Sync_flag sync)
```

```
int MPIRT::Handler::Set_Sync (MPIRT::Sync_Flag sync)
```

⊤ (final)

MPIRT_HANDLER_SET_QOS (handler, handler_qos)

INOUT	handler	handler (ref. to MPIRT_HANDLER)
IN	handler_qos	QoS for the handler (MPIRT_HANDLER_QOS)

```
int MPIRT_Handler_set_qos (MPIRT_Handler *handler, MPIRT_Qos_Handler handler_qos)
```

```
int MPIRT::Handler::Set_Qos (const MPIRT::Qos_Handler &handler_qos)
```

MPIRT_HANDLER_SET_EXTRA_STATE (handler, extra_state)

INOUT	handler	handler (ref. to MPIRT_HANDLER)
IN	extra_state	user-supplied parameters for the handler (MPIRT_ADDRESS)

```
int MPIRT_Handler_set_extra_state (MPIRT_Handler *handler,
            MPIRT_Address extra_state)
```

```
int MPIRT::Handler::Set_Extra_State (const MPIRT::Address extra_state)
```

⊥ (final2)
⊤ (final)
The duplication, free, and equality testing operations and the get and set name parameter operations for handlers are inherited from the general class of MPIRT_OBJECT (see Section 2.2).
⊥ (final)

Advice to users. Duplication can be used as the mechanism by which error handlers are shared between multiple channels or other objects. (*End of advice to users.*)

4.4 Summary

This section summarizes enumerations and type declarations that are introduced in this chapter.

Miscellaneous Definitions		
Item	*What is it*	*Defined as:*
MPIRT_TRIGGER	trigger object	trigger object
MPIRT_RECEPTOR	receptor object	receptor object
MPIRT_HANDLER	Handler object	Handler object
MPIRT_OBJECT_TYPE_TRIGGER	a trigger's MPIRT_OBJECT_TYPE	enumeration
MPIRT_OBJECT_TYPE_RECEPTOR	a trigger's MPIRT_OBJECT_TYPE	enumeration
MPIRT_OBJECT_TYPE_HANDLER	a handler's MPIRT_OBJECT_TYPE	enumeration
MPIRT_TRIGGER_NULL	null trigger	null trigger object
MPIRT_RECEPTOR_NULL	null receptor	null receptor object
MPIRT_HANDLER_NULL	null handler	null handler object
MPIRT_ERR_RECEPTOR_ALREADY_EXISTS	error code for create function	enumeration
MPIRT_ERR_NO_RECEPTOR_EXISTS	error code for retrieve function	enumeration
`MPIRT_HANDLER_FN` `MPIRT::HANDLER::FN`	handler function prototype definition	type definition
MPIRT_PRIORITY_INHERIT	handler QoS priority parameter for synchronous handlers	enumeration
MPIRT_CONDITION_UNKNOWN	value for parameter condition given to remote synchronous handlers	enumeration (MPIRT_CONDITION)
MPIRT_HANDLER_FN_LW_MASK_EXTRA_INFO	structure given as extra_info for MPIRT_BITWISE_INT light-weight handler function	structure

⊤ (final2)
⊥ (final2)

Handler Function Return Codes		
RETURN VALUE	*SYNCHRONOUS*	*ASYNCHRONOUS*
MPIRT_CONTINUE	processing continues to next synchronous handler	ignored
MPIRT_SUCCESS	stops further synchronous handler processing, enables a retry of the operation if local to trigger, and raises MPIRT_HANDLER_COND_SUCCESS	raises MPIRT_HANDLER_COND_SUCCESS
MPIRT_FAILURE	stops further synchronous handler processing and raises MPIRT_HANDLER_COND_FAILURE	raises MPIRT_HANDLER_COND_FAILURE

Conditions			
Condition Enumeration	*When Raised*	*Name String*	*Recoverable*
MPIRT_TRIGGER_COND_EVENT_RAISED	upon raising a trigger	"RAISED"	no
MPIRT_RECEPTOR_COND_EVENT_ARRIVED	upon activation of a receptor	"ARRIVED"	no
MPIRT_HANDLER_COND_SUCCESS	successful handler completion	"SUCCESS"	no
MPIRT_HANDLER_COND_FAILURE	failed handler completion	"FAILURE"	no
MPIRT_HANDLER_COND_QOS_FAILURE	handler QOS failure	"QOS"	no

Handler Processing Modes (MPIRT_SYNC_FLAG values)	
Mode	*Method*
MPIRT_HANDLER_SYNC	handler processed synchronously
MPIRT_HANDLER_ASYNC	handler processed asynchronously

Light Weight Handler Functions		
C *Enumeration* C++ *Enumeration*	*Operation*	*extra_state*
MPIRT_HANDLER_FN_LW_SET_INT MPIRT::Handler::FN_LW_SET_INT	sets value to 1	pointer to `integer`
MPIRT_HANDLER_FN_LW_CLEAR_INT MPIRT::Handler::FN_LW_CLEAR_INT	sets value to 0	
MPIRT_HANDLER_FN_LW_TOGGLE_INT MPIRT::Handler::FN_LW_TOGGLE_INT	Logical *not* of value	
MPIRT_HANDLER_FN_LW_COMPLEMENT_INT MPIRT::Handler::FN_LW_COMPLEMENT_INT	Binary *not* of value	
MPIRT_HANDLER_FN_LW_INCREMENT_INT MPIRT::Handler::FN_LW_INCREMENT_INT	adds 1 to the value	
MPIRT_HANDLER_FN_LW_DECREMENT_INT MPIRT::Handler::FN_LW_DECREMENT_INT	subtracts 1 from the value	
MPIRT_HANDLER_FN_LW_BITWISE_INT MPIRT::Handler::FN_LW_BITWISE_INT	Bitwise *and*, *or*, and *xor*	pointer to structure MPIRT_HANDLER_FN_LW_MASK_EXTRA_INFO

Event Delivery Abstraction Operations		
Language Independent Specification	C	C++
MPIRT_EVDEL_GET_EVENT_NAME	MPIRT_Evdel_get_event_name	MPIRT::Evdel::Get_Event_Name
MPIRT_EVDEL_GET_SCOPE	MPIRT_Evdel_get_scope	MPIRT::Evdel::Get_Scope
MPIRT_EVDEL_SET_EVENT_NAME	MPIRT_Evdel_set_event_name	MPIRT::Evdel::Set_Event_Name
MPIRT_EVDEL_SET_SCOPE	MPIRT_Evdel_set_scope	MPIRT::Evdel::Set_Scope

Trigger Specific Operations		
Language Independent Specification	C	C++
MPIRT_TRIGGER_GET_EVENT_NAME	MPIRT_Trigger_get_event_name	MPIRT::Trigger::Get_Event_Name
MPIRT_TRIGGER_GET_SCOPE	MPIRT_Trigger_get_scope	MPIRT::Trigger::Get_Scope
MPIRT_TRIGGER_SET_EVENT_NAME	MPIRT_Trigger_set_event_name	MPIRT::Trigger::Set_Event_Name
MPIRT_TRIGGER_SET_SCOPE	MPIRT_Trigger_set_scope	MPIRT::Trigger::Set_Scope
MPIRT_TRIGGER_CREATE	MPIRT_Trigger_create	MPIRT::Trigger
MPIRT_TRIGGER_GET_QOS	MPIRT_Trigger_get_qos	MPIRT::Trigger::Get_Qos
MPIRT_TRIGGER_SET_QOS	MPIRT_Trigger_set_qos	MPIRT::Trigger::Set_Qos
MPIRT_TRIGGER_RAISE	MPIRT_Trigger_raise	MPIRT::Trigger::Raise
MPIRT_TRIGGER_GET_NAME	MPIRT_Trigger_get_name	MPIRT::Trigger::Get_Name
MPIRT_TRIGGER_SET_NAME	MPIRT_Trigger_set_name	MPIRT::Trigger::Set_Name
MPIRT_TRIGGER_DUP	MPIRT_Trigger_dup	MPIRT::Trigger MPIRT::Trigger::Dup MPIRT::Trigger::operator=
MPIRT_TRIGGER_FREE	MPIRT_Trigger_free	MPIRT::~Trigger MPIRT::Trigger::Free
MPIRT_TRIGGER_IS_EQUAL	MPIRT_Trigger_is_equal	MPIRT::Trigger::Is_equal
MPIRT_TRIGGER_ATTACH_ATTR	MPIRT_Trigger_attach_attr	MPIRT::Trigger::Attach_Attr
MPIRT_TRIGGER_RETRIEVE_ATTR	MPIRT_Trigger_retrieve_attr	MPIRT::Trigger::Retrieve_Attr
MPIRT_TRIGGER_DELETE_ATTR	MPIRT_Trigger_delete_attr	MPIRT::Trigger::Delete_Attr

Receptor Specific Operations		
Language Independent Specification	C	C++
MPIRT_RECEPTOR_GET_EVENT_NAME	MPIRT_Receptor_get_event_name	MPIRT::Receptor::Get_Event_Name
MPIRT_RECEPTOR_GET_SCOPE	MPIRT_Receptor_get_scope	MPIRT::Receptor::Get_Scope
MPIRT_RECEPTOR_SET_EVENT_NAME	MPIRT_Receptor_set_event_name	MPIRT::Receptor::Set_Event_Name
MPIRT_RECEPTOR_SET_SCOPE	MPIRT_Receptor_set_scope	MPIRT::Receptor::Set_Scope
MPIRT_RECEPTOR_CREATE	MPIRT_Receptor_create	MPIRT::Receptor
MPIRT_RECEPTOR_GET_QOS	MPIRT_Receptor_get_qos	MPIRT::Receptor::Get_Qos
MPIRT_RECEPTOR_GET_INITIAL_STATE	MPIRT_Receptor_get_initial_state	MPIRT::Receptor::Get_Initial_State
MPIRT_RECEPTOR_GET_HANDLERS	MPIRT_Receptor_get_handlers	MPIRT::Receptor::Get_Handlers
MPIRT_RECEPTOR_SET_QOS	MPIRT_Receptor_set_qos	MPIRT::Receptor::Set_Qos
MPIRT_RECEPTOR_SET_INITIAL_STATE	MPIRT_Receptor_set_initial_state	MPIRT::Receptor::Set_Initial_State
MPIRT_RECEPTOR_SET_HANDLERS	MPIRT_Receptor_set_handlers	MPIRT::Receptor::Set_Handlers
MPIRT_RETRIEVE_RECEPTOR	MPIRT_Retrieve_receptor	MPIRT::Retrieve_Receptor
MPIRT_RECEPTOR_IS_ENABLED	MPIRT_Receptor_is_enabled	MPIRT::Receptor::Is_Enabled
MPIRT_RECEPTOR_ENABLE	MPIRT_Receptor_enable	MPIRT::Receptor::Enable
MPIRT_RECEPTOR_DISABLE	MPIRT_Receptor_disable	MPIRT::Receptor::Disable
MPIRT_RECEPTOR_GET_NAME	MPIRT_Receptor_get_name	MPIRT::Receptor::Get_Name
MPIRT_RECEPTOR_SET_NAME	MPIRT_Receptor_set_name	MPIRT::Receptor::Set_Name
MPIRT_RECEPTOR_DUP	MPIRT_Receptor_dup	MPIRT::Receptor MPIRT::Receptor::Dup MPIRT::Receptor::operator=
MPIRT_RECEPTOR_FREE	MPIRT_Receptor_free	MPIRT::∼Receptor MPIRT::Receptor::Free
MPIRT_RECEPTOR_IS_EQUAL	MPIRT_Receptor_is_equal	MPIRT::Receptor::Is_equal
MPIRT_RECEPTOR_ATTACH_ATTR	MPIRT_Receptor_attach_attr	MPIRT::Receptor::Attach_Attr
MPIRT_RECEPTOR_RETRIEVE_ATTR	MPIRT_Receptor_retrieve_attr	MPIRT::Receptor::Retrieve_Attr
MPIRT_RECEPTOR_DELETE_ATTR	MPIRT_Receptor_delete_attr	MPIRT::Receptor::Delete_Attr

Handler Specific Operations		
Language Independent Specification	C	C++
MPIRT_HANDLER_CREATE	MPIRT_Handler_create	MPIRT::Handler
MPIRT_HANDLER_GET_HANDLER_FN	MPIRT_Handler_get_handler_fn	MPIRT::Handler::Get_Handler_Fn
MPIRT_HANDLER_GET_QOS	MPIRT_Handler_get_qos	MPIRT::Handler::Get_Qos
MPIRT_HANDLER_GET_SYNC	MPIRT_Handler_get_sync	MPIRT::Handler::Get_Sync
MPIRT_HANDLER_GET_EXTRA_STATE	MPIRT_Handler_get_extra_state	MPIRT::Handler::Get_Extra_State
MPIRT_HANDLER_SET_HANDLER_FN	MPIRT_Handler_set_handler_fn	MPIRT::Handler::Set_Handler_Fn
MPIRT_HANDLER_SET_QOS	MPIRT_Handler_set_qos	MPIRT::Handler::Set_Qos
MPIRT_HANDLER_SET_SYNC	MPIRT_Handler_set_sync	MPIRT::Handler::Set_Sync
MPIRT_HANDLER_SET_EXTRA_STATE	MPIRT_Handler_set_extra_state	MPIRT::Handler::Set_Extra_State
MPIRT_HANDLER_GET_NAME	MPIRT_Handler_get_name	MPIRT::Handler::Get_Name
MPIRT_HANDLER_SET_NAME	MPIRT_Handler_set_name	MPIRT::Handler::Set_Name
MPIRT_HANDLER_DUP	MPIRT_Handler_dup	MPIRT::Handler MPIRT::Handler::Dup MPIRT::Handler::operator=
MPIRT_HANDLER_FREE	MPIRT_Handler_free	MPIRT::~Handler MPIRT::Handler::Free
MPIRT_HANDLER_IS_EQUAL	MPIRT_Handler_is_equal	MPIRT::Handler::Is_equal
MPIRT_HANDLER_ATTACH_ATTR	MPIRT_Handler_attach_attr	MPIRT::Handler::Attach_Attr
MPIRT_HANDLER_RETRIEVE_ATTR	MPIRT_Handler_retrieve_attr	MPIRT::Handler::Retrieve_Attr
MPIRT_HANDLER_DELETE_ATTR	MPIRT_Handler_delete_attr	MPIRT::Handler::Delete_Attr

Miscellaneous Bindings		
Item	*C Binding*	*C++ Binding*
MPIRT_TRIGGER	MPIRT_Trigger	MPIRT::Trigger
MPIRT_RECEPTOR	MPIRT_Receptor	MPIRT::Receptor
MPIRT_HANDLER	MPIRT_Handler	MPIRT::Handler
MPIRT_OBJECT_TYPE_TRIGGER	MPIRT_OBJECT_TYPE_TRIGGER	MPIRT::OBJECT_TYPE_TRIGGER
MPIRT_OBJECT_TYPE_RECEPTOR	MPIRT_OBJECT_TYPE_RECEPTOR	MPIRT::OBJECT_TYPE_RECEPTOR
MPIRT_OBJECT_TYPE_HANDLER	MPIRT_OBJECT_TYPE_HANDLER	MPIRT::OBJECT_TYPE_HANDLER
MPIRT_TRIGGER_NULL	MPIRT_TRIGGER_NULL	MPIRT::TRIGGER_NULL
MPIRT_RECEPTOR_NULL	MPIRT_RECEPTOR_NULL	MPIRT::RECEPTOR_NULL
MPIRT_HANDLER_NULL	MPIRT_HANDLER_NULL	MPIRT::HANDLER_NULL
MPIRT_ERR_RECEPTOR_ALREADY_EXISTS	MPIRT_ERR_RECEPTOR_ALREADY_EXISTS	MPIRT::ERR_RECEPTOR_ALREADY_EXISTS
MPIRT_ERR_NO_RECEPTOR_EXISTS	MPIRT_ERR_NO_RECEPTOR_EXISTS	MPIRT::ERR_NO_RECEPTOR_EXISTS
MPIRT_HANDLER_FN	MPIRT_HANDLER_FN	MPIRT::Handler::FN
MPIRT_PRIORITY_INHERIT	MPIRT_PRIORITY_INHERIT	MPIRT::PRIORITY_INHERIT
MPIRT_CONDITION_UNKNOWN	MPIRT_CONDITION_UNKNOWN	MPIRT::CONDITION_UNKNOWN
MPIRT_HANDLER_FN_LW_MASK_EXTRA_INFO	MPIRT_HANDLER_FN_LW_MASK_EXTRA_INFO	MPIRT::Handler::FN_LW_MASK_EXTRA_INFO
MPIRT_TRIGGER_COND_EVENT_RAISED	MPIRT_TRIGGER_COND_EVENT_RAISED	MPIRT::Trigger::COND_EVENT_RAISED
MPIRT_RECEPTOR_COND_EVENT_ARRIVED	MPIRT_RECEPTOR_COND_EVENT_ARRIVED	MPIRT::Receptor::COND_EVENT_ARRIVED
MPIRT_HANDLER_COND_SUCCESS	MPIRT_HANDLER_COND_SUCCESS	MPIRT::Handler::COND_SUCCESS
MPIRT_HANDLER_COND_FAILURE	MPIRT_HANDLER_COND_FAILURE	MPIRT::Handler::COND_FAILURE
MPIRT_HANDLER_COND_QOS_FAILURE	MPIRT_HANDLER_COND_QOS_FAILURE	MPIRT::Handler::COND_QOS_FAILURE
MPIRT_HANDLER_SYNC	MPIRT_HANDLER_SYNC	MPIRT::Handler::SYNC
MPIRT_HANDLER_ASYNC	MPIRT_HANDLER_ASYNC	MPIRT::Handler::ASYNC
MPIRT_SUCCESS	MPIRT_SUCCESS	MPIRT::SUCCESS
MPIRT_FAILURE	MPIRT_FAILURE	MPIRT::FAILURE
MPIRT_CONTINUE	MPIRT_CONTINUE	MPIRT::CONTINUE
MPIRT_HANDLER_FN_LW_SET_INT	MPIRT_HANDLER_FN_LW_SET_INT	MPIRT::Handler::FN_LW_SET_INT
MPIRT_HANDLER_FN_LW_CLEAR_INT	MPIRT_HANDLER_FN_LW_CLEAR_INT	MPIRT::Handler::FN_LW_CLEAR_INT
MPIRT_HANDLER_FN_LW_TOGGLE_INT	MPIRT_HANDLER_FN_LW_TOGGLE_INT	MPIRT::Handler::FN_LW_TOGGLE_INT
MPIRT_HANDLER_FN_LW_COMPLEMENT_INT	MPIRT_HANDLER_FN_LW_COMPLEMENT_INT	MPIRT::Handler::FN_LW_COMPLEMENT_INT
MPIRT_HANDLER_FN_LW_INCREMENT_INT	MPIRT_HANDLER_FN_LW_INCREMENT_INT	MPIRT::Handler::FN_LW_INCREMENT_INT
MPIRT_HANDLER_FN_LW_DECREMENT_INT	MPIRT_HANDLER_FN_LW_DECREMENT_INT	MPIRT::Handler::FN_LW_DECREMENT_INT
MPIRT_HANDLER_FN_LW_BITWISE_INT	MPIRT_HANDLER_FN_LW_BITWISE_INT	MPIRT::Handler::FN_LW_BITWISE_INT

Chapter 5

Buffer Management

Current Status: First vote passed 8/0/1 on 5/20/98.
Second vote passed 10/0/2 on 6/23/98. Third vote passed 9/0/0 on 12/3/98.

5.1 Introduction

This section describes an interface for buffer management. Buffers of uniform size and dataspecs are organized into collections using *V-containers*. These containers with buffers pass into buffer iterators to be used by channels for message transfers. The order of the buffers within the buffer iterator is specified by the user, and it defines the order in which buffers are consumed from and returned to a channel endpoint and an application. The functionality for inserting a buffer and removing a buffer from a buffer iterator is also provided. This functionality allows application and implementation to fill a buffer with a message or retrieve a message from a buffer. Figures 5.1 and 5.2 show the relationship of the buffers and buffer iterators to message transfer operations on channels. Figures 5.3 and 5.4 illustrate linkages of input and output buffer iterators of two channels; Figure 5.3 provides the user with the message before forwarding, while Figure 5.4 shortcuts the message through, never handing the message off to the user before shipping it out again.

Following MPI/RT's object-oriented methodology with deferred early binding, the parameters for buffers and buffer iterators are specified up front. Resources are allocated only when these objects are committed (see Chapter 12). The only exceptions to this general MPI/RT rule are buffers whose memory can be allocated by an application. Once an MPI/RT object has been committed, its object descriptor can no longer be modified. This includes the descriptors for committed buffers and buffer iterators.

5.2 Buffer Object

This section defines operations for creation and manipulation of a buffer object.

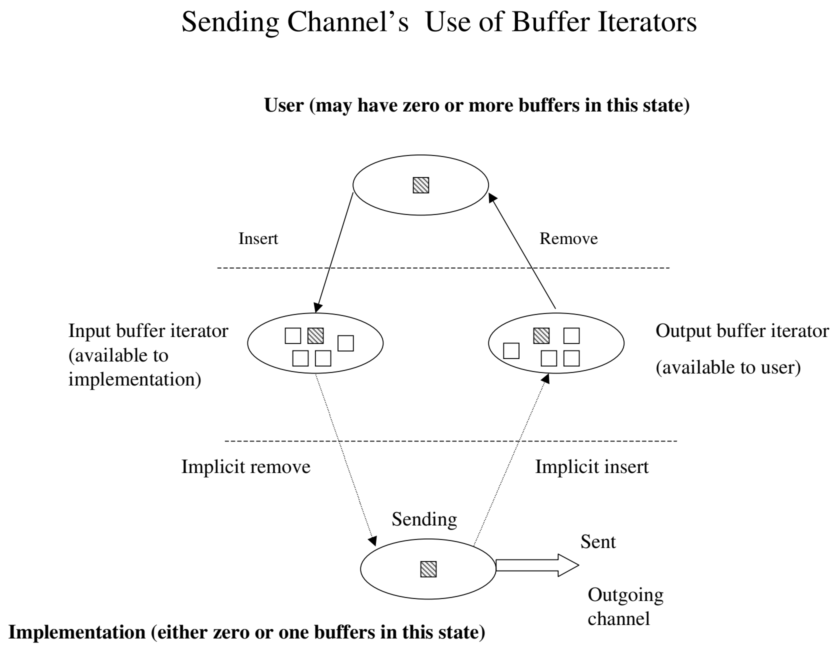

Figure 5.1: Send side buffer management diagram. Full buffers go into the input iterator, and empty buffers come out of the output iterator.

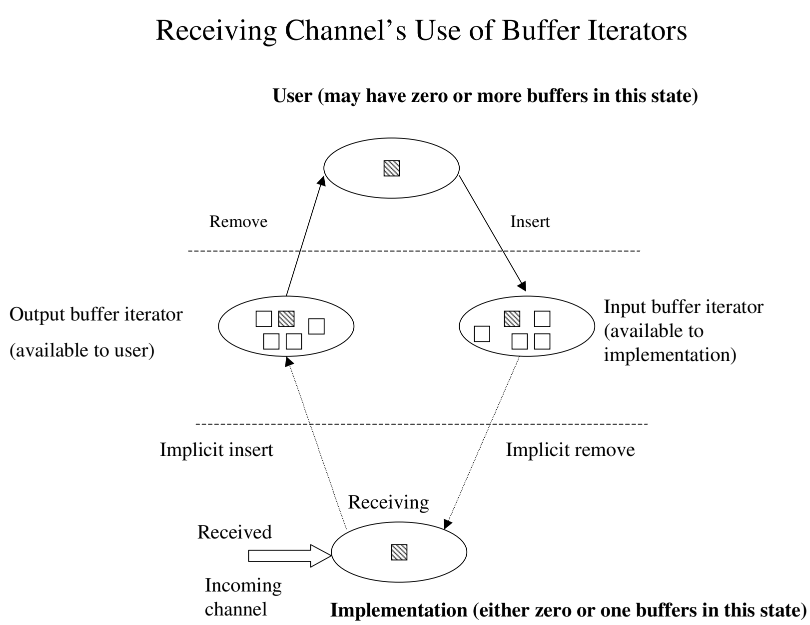

Figure 5.2: Receive side buffer management diagram. Full buffers come out of the output iterator, and empty buffers go into the input iterator.

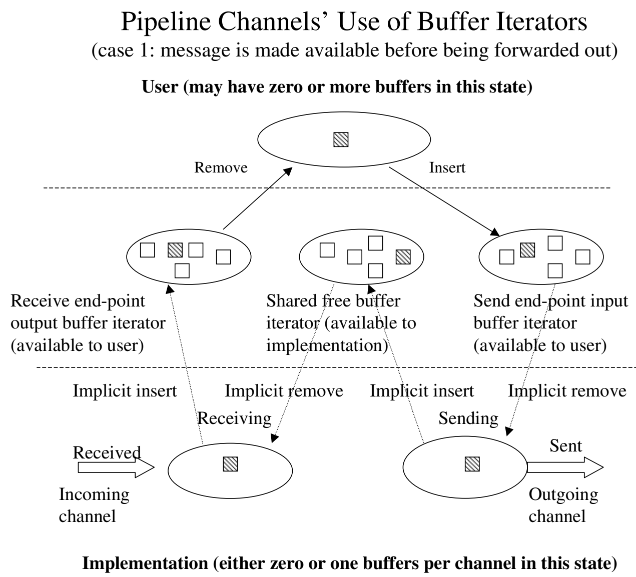

Figure 5.3: A two-channel example where the user gets the buffer after receipt on the receive-side channel and then forwards the buffer to the send-side channel. The channels are unified by making the output iterator of the input channel the same as the input iterator of the output channel.

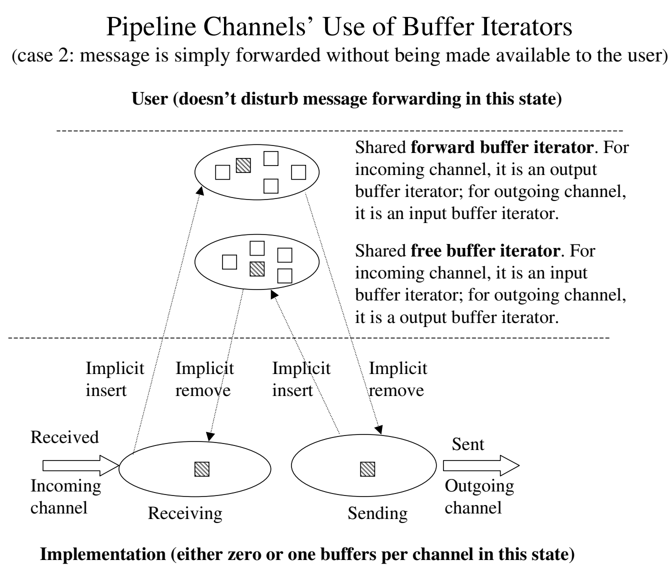

Figure 5.4: A two-channel example where the user never gets the buffer after receipt on the receive-side channel because it is directly forwarded within the implementation to the send-side channel, creating "software cutthrough." The channels are unified by making the output iterator of the input channel the same as the input iterator of the output channel.

MPIRT_BUFFER_CREATE (name, bufsize, dataspec, base, buffer)

IN	name	name of object (MPIRT_STRING_NAME)
IN	bufsize	number of elements in the buffer (non-negative integer)
IN	dataspec	dataspec of elements in the buffer (MPIRT_DATASPEC)
IN	base	MPIRT_SYSTEM_MEM_ALLOC for the system to allocate memory when the buffer is committed, or an address of the user-allocated buffer (MPIRT_ADDRESS)
OUT	buffer	buffer (ref. to MPIRT_BUFFER)

```
int MPIRT_Buffer_create (MPIRT_String_name name, int bufsize,
            MPIRT_Dataspec dataspec, MPIRT_Address base, MPIRT_Buffer *buffer)
```

```
MPIRT::Buffer (const MPIRT::String_Name name, int bufsize, const
            MPIRT::Dataspec &dataspec, MPIRT::Address base)
```

⊤ (final)

A buffer object is an abstraction of memory that can contain a message or several messages to allow synchronization of the access to the memory between a user and an implementation. The following rules for a buffer describes when it is safe to read and write into a buffer memory to ensure than the data is not corrupted:

- A buffer object is available if it is committed until it is inserted into a buffer iterator, and after it is removed from a buffer iterator until it is inserted again.
- It is illegal to read and write to a memory location after commit unless every committed buffer object whose memory region contains that memory location is available.
- It is illegal to insert a buffer into a buffer iterator unless the buffer is available.
- It is illegal for a buffer object to be in the initial buffer list of more than one buffer iterator being committed.
- An implementation will not read or write to a memory location unless at least one user-created buffer containing that memory location is inserted into a buffer iterator.

Figure 5.5 presents the state transition diagram for a buffer and depicts when a buffer is available.

⊥ (final)

A buffer object has one parameter, label, which is not part of the object description. There are two operations on buffer labels defined in Section 5.3. Buffer labels are used in buffer iterators with ⊤ (final2) policy of MPIRT_BUFITER_SORTED or MPIRT_BUFITER_SORTED_MAPPED to define the order in which ⊥ (final2) they are retrieved (see Section 5.4), but may also be used by the programmer as an informational value. Initially, when a buffer is created, it has a default label MPIRT_BUFFER_LABEL_DEFAULT. Labels are not transmitted to other processes as a part of the message. They are local to the process.

Users are allowed to create zero-length buffers. An implementation is allowed to ignore the dataspec and base for buffers that are zero length.

The next three operations query the parameters of the buffer object.

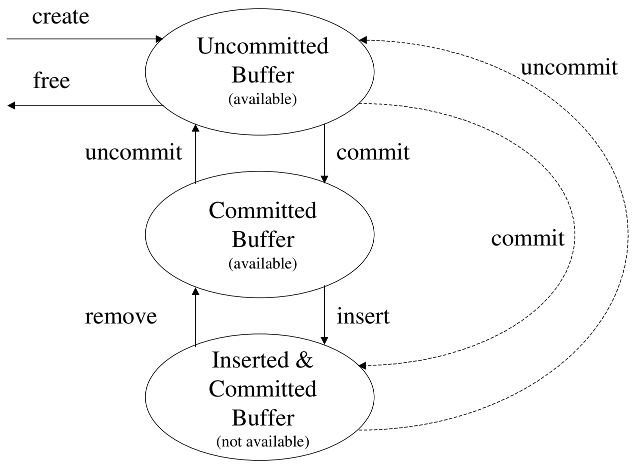

Figure 5.5: Buffer state transition diagram. The dashed lines indicate that a buffer in a buffer iterator's initial list is automatically inserted and removed from the buffer iterator during commit and uncommit operations, respectively, on the buffer iterator.

MPIRT_BUFFER_GET_BUFSIZE (buffer, bufsize)

IN	buffer	buffer (MPIRT_BUFFER)
OUT	bufsize	number of elements in the buffer (ref. to non-negative integer)

```
int MPIRT_Buffer_get_bufsize (MPIRT_Buffer buffer, int *bufsize)
```

```
int MPIRT::Buffer::Get_Bufsize (int &bufsize) const
```

MPIRT_BUFFER_GET_DATASPEC (buffer, dataspec)

IN	buffer	buffer (MPIRT_BUFFER)
OUT	dataspec	dataspec of each element in the buffer (ref. to MPIRT_DATASPEC)

```
int MPIRT_Buffer_get_dataspec (MPIRT_Buffer buffer, MPIRT_Dataspec *dataspec)
```

```
int MPIRT::Buffer::Get_Dataspec (MPIRT::Dataspec *&dataspec) const
```

MPIRT_BUFFER_GET_BASE (buffer, base)

IN	buffer	buffer (MPIRT_BUFFER)
OUT	base	MPIRT_SYSTEM_MEM_ALLOC for the system to allocate memory before commit, or an address of a user-allocated buffer before and after commit, or a system-allocated buffer after commit (ref. to MPIRT_ADDRESS)

```
int MPIRT_Buffer_get_base (MPIRT_Buffer buffer, MPIRT_Address *base)
```

```
int MPIRT::Buffer::Get_Base (MPIRT::Address &base) const
```

The next three operations set the parameters of the buffer object.

MPIRT_BUFFER_SET_BUFSIZE (buffer, bufsize)

INOUT	buffer	buffer (ref. to MPIRT_BUFFER)
IN	bufsize	number of elements in the buffer (non-negative integer)

```
int MPIRT_Buffer_set_bufsize (MPIRT_Buffer *buffer, int bufsize)
```

```
int MPIRT::Buffer::Set_Bufsize (int bufsize)
```

MPIRT_BUFFER_SET_DATASPEC(buffer, dataspec)

INOUT	buffer	buffer (ref. to MPIRT_BUFFER)
IN	dataspec	dataspec of each element in the buffer (MPIRT_DATASPEC)

```
int MPIRT_Buffer_set_dataspec (MPIRT_Buffer *buffer, MPIRT_Dataspec dataspec)
```

```
int MPIRT::Buffer::Set_Dataspec (const MPIRT::Dataspec &dataspec)
```

MPIRT_BUFFER_SET_BASE (buffer, base)

INOUT	buffer	buffer (ref. to MPIRT_BUFFER)
IN	base	MPIRT_SYSTEM_MEM_ALLOC for the system to allocate memory when the buffer is committed, or an address of a user-allocated buffer (MPIRT_ADDRESS)

```
int MPIRT_Buffer_set_base (MPIRT_Buffer *buffer, MPIRT_Address base)
```

```
int MPIRT::Buffer::Set_Base (MPIRT::Address base)
```

⊤ (final)

If a buffer is created using MPIRT_SYSTEM_MEM_ALLOC (call it buf1) and then duplicated (call it buf2), then both buffers will refer to the same system allocated memory location. However, if there is *any* call to MPIRT_BUFFER_SET_BASE for either buffer, then they refer to different memory, even if the base is set back to MPIRT_SYSTEM_MEM_ALLOC for both. Essentially, this means that after a buffer with base MPIRT_SYSTEM_MEM_ALLOC has been duplicated, any subsequent call to MPIRT_BUFFER_SET_BASE severs any relationship between the original and the duplicate.

Rationale. Since each buffer created using MPIRT_SYSTEM_MEM_ALLOC refers to a different memory location, then an implementation must have it's own internal scheme to keep track of how many different system allocated regions it must create at commit time so that it can fill in the base address of each such buffer during commit. Simple duplication of any such buffer means that the original and duplicate share the same base address. Once the relationship is severed with a call to MPIRT_BUFFER_SET_BASE, there is no convenient way to reattach to a specific system allocated memory region without adding more complexity to the interface. (*End of rationale.*)

⊥ (final)

The duplication, free, and comparison operations and the get and set name parameter operations for buffers are inherited from the general class of MPIRT_OBJECT (see Section 2.2).

5.3 Buffer Operations

This section defines operations on the buffer object.

5.3.1 Operations on Buffer Labels

The operations in this subsection manipulate the buffer label parameter.

The following routine assigns the buffer label.

MPIRT_BUFFER_ASSIGN_LABEL (buffer, label)

INOUT	buffer	buffer (ref. to MPIRT_BUFFER)
IN	label	buffer label (integer)

```
int MPIRT_Buffer_assign_label (MPIRT_Buffer *buffer, int label)
```

```
int MPIRT::Buffer::Assign_Label (int label)
```

The following routine retrieves the buffer label.

MPIRT_BUFFER_RETRIEVE_LABEL (buffer, label)

IN	buffer	buffer (MPIRT_BUFFER)
OUT	label	buffer label (ref. to integer)

```
int MPIRT_Buffer_retrieve_label (MPIRT_Buffer buffer, int *label)
```

```
int MPIRT::Buffer::Retrieve_Label (int &label) const
```

5.3.2 Buffer-Partitioning Operations

In order to group together buffers of the same dataspec and size so any of them can be used for a buffer iterator (for a channel), users can create a *V-container* with buffers in it. Several convenience operations are introduced here. One allows users to create a *V-container* with several buffers of the same dataspec and bufsize from a larger, previously created buffer (see Figure 5.6). Another allows users to create a buffer from an already created one of the specified size that starts at the specified location of the existing buffer (see Figure 5.7). All of the created buffers use the same memory as the original buffer from which they are created.

The operation MPIRT_BUFFER_CREATE_SUBBUFFERS allows users to split a large buffer into the requested number of smaller, non-overlapping buffers of requested size and of the same dataspec as the large buffer. The newly created buffers are returned in a *V-container* of buffers, and the remaining part of the large buffer is also returned as a buffer. The argument for the remaining part of the large buffer, rembuffer, is set to MPIRT_BUFFER_NULL if the large buffer is evenly divisible. The new buffers form a partition of the large buffer, starting from its base into buffers_count buffers of size bufsize, and are put into a *V-container*. The names of the buffers in the created *V-container* are the name of the *V-container* followed by the number. The number is the index of the buffer in the *V-container* at the time of creation. For example, the first created buffer in the *V-container* named *bufcont* has the name *bufcont1*.

MPIRT_BUFFER_CREATE_SUBBUFFERS (inbuffer, buffers_count, bufsize, bufcontainer_name, bufcontainer, rembuffer_name, rembuffer)

IN	inbuffer	the buffer from which buffers are created (MPIRT_BUFFER)
IN	buffers_count	number of buffers to create in bufcontainer (integer)
IN	bufsize	number of elements of inbuffer dataspec in each buffer in bufcontainer (non-negative integer)
IN	bufcontainer_name	the name of bufcontainer (MPIRT_STRING_NAME)
OUT	bufcontainer	*V-container* with the created buffers (ref. to MPIRT_CVECTOR)
IN	rembuffer_name	the name of the buffer of the remaining inbuffer part (MPIRT_STRING_NAME)
OUT	rembuffer	buffer with the remaining part of the inbuffer (ref. to MPIRT_BUFFER)

```
int MPIRT_Buffer_create_subbuffers (MPIRT_Buffer inbuffer, int buffers_count,
             int bufsize, MPIRT_String_name container_name,
             MPIRT_Cvector *bufcontainer, MPIRT_String_name rembuffer_name,
             MPIRT_Buffer *rembuffer)
```

```
MPIRT::Cvector (const MPIRT::Buffer &inbuffer, int buffers_count, int bufsize,
             const MPIRT::String_Name container_name, const
             MPIRT::String_Name rembuffer_name, MPIRT::Buffer *&rembuffer)
```

Rationale. The rationale for creating a vector container instead of a set container is that the user can directly pass this container into the MPIRT_BUFITER_CREATE operation.

This operation allows users as well as implementors to allocate a large piece of memory and then carve it up as needed for the channels. This improves performance predictability by reducing overhead of memory management by reducing the number of *malloc* and *free* operations required to create and destroy numbers of buffers.

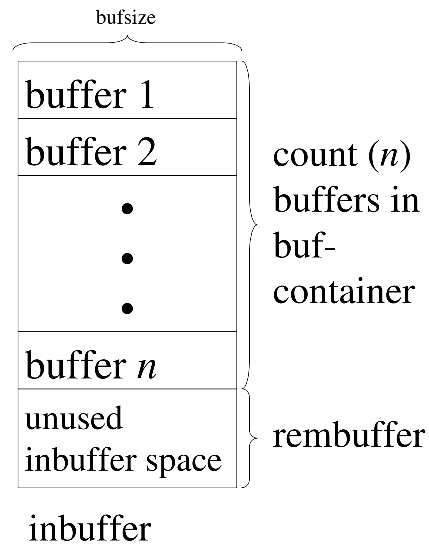

Figure 5.6: A single buffer inbuffer is partitioned into n bufsize-sized buffers, leaving rembuffer unused space in inbuffer.

Also, the ability to carve a large buffer into an ordered set of the smaller ones provides a simple way to allocate buffers for overlapping channels that support data fusion and data dispersal. It is especially suited to support an implementation of collective channels like scatter and gather. (*End of rationale.*)

Advice to users. When users create a buffer to be carved up for the use of channel buffer iterators, they should use a special memory that system uses for data transfers. This memory may be platform specific.

While being able to create several analogous buffers in one operation, users still need to determine the actual buffers in the container. This can be achieved by traversing the entire container using MPIRT_CVECTOR_RETRIEVE, which returns a reference to one buffer at a time.

Users are responsible for destroying the created buffers and the *V-container* created by this call. (*End of advice to users.*)

MPIRT_BUFFER_CREATE_SUBBUFFER (inbuffer, displacement, bufsize, outbuffer_name, outbuffer)

IN	inbuffer	the buffer to be partitioned (MPIRT_BUFFER)
IN	displacement	the number of elements from the base of the inbuffer that defines the start position of the created buffer (non-negative integer)
IN	bufsize	number of elements of inbuffer dataspec in the created buffer (non-negative integer)
IN	outbuffer_name	the name of the created buffer (MPIRT_STRING_NAME)
OUT	outbuffer	created buffer (ref. to MPIRT_BUFFER)

```
int MPIRT_Buffer_create_subbuffer (MPIRT_Buffer inbuffer, int displacement,
             int bufsize, MPIRT_String_name outbuffer_name,
             MPIRT_Buffer *outbuffer)
```

```
MPIRT::Buffer (const MPIRT::Buffer &inbuffer, int displacement, int bufsize,
             const MPIRT::String_Name outbuffer_name)
```

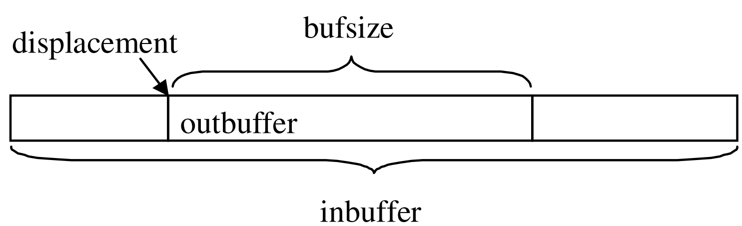

Figure 5.7: A bufsize-sized subset is created within inbuffer, a larger buffer.

Rationale. This operation allows users to carve a larger buffer into various-sized other buffers or even overlapping buffers by carving the inbuffer multiple times. (*End of rationale.*)

5.4 Buffer Iterator

A buffer iterator is an object for managing a collection of buffers. In particular, the buffer iterator construct provides a mechanism for ordering buffers with respect to their consumption by and return to a channel endpoint. Each point-to-point channel endpoint has two iterators: one for input, from which buffers are taken to be sent or received, and another for output, into which buffers are put after a channel operation completes. Collective channel endpoints have more than one pair of in and out buffer iterators whenever an endpoint is involved in both sending and receiving of messages.

The following operation creates a buffer iterator which can contain up to the specified maximum number of buffers and initially contains the buffers passed by the *V-container* of buffers. The policy defines the way in which the buffers are ordered with respect to the iterator accessor functions, MPIRT_BUFITER_REMOVE and MPIRT_BUFITER_INSERT, defined below.

Labels are provided in order to allow buffers from the *V-container* to be ordered by the user for the purpose of controlling the queuing order of the buffers. The buffer labels can be assigned and retrieved by using MPIRT_BUFFER_ASSIGN_LABEL and MPIRT_BUFFER_RETRIEVE_LABEL (see Section 5.3) and are used by the implementation for a buffer iterator that is using a policy of MPIRT_BUFITER_SORTED.

> *Advice to users.* A buffer object can be a member of at most one buffer iterator at any given time. Since the base address of a physical memory buffer can be shared by more than one buffer object, the memory can be referenced by more than one buffer object and can, therefore, be used in multiple buffer iterators. The use of the function MPIRT_OBJECT_DUP on a buffer object can be useful to achieve this.
>
> A buffer object can be a member of any number of containers at any given time. However, a buffer can belong, at most, to one buffer iterator and/or channel at any given time. The memory used by a buffer can be referenced by multiple buffer objects simultaneously. (*End of advice to users.*)

> *Advice to implementors.* A buffer object can be a member of at most one buffer iterator at any given time. This allows optimizations to be made for buffer iterators in the storage in the buffer iterator (for example, a linked list using a field inside the buffer object data structure itself). However, a buffer object can be a member of any number of containers at any given time. (*End of advice to implementors.*)

⊤ (final)

MPIRT_BUFITER_CREATE (name, maxbufcount, bufsize, dataspec, allowed_buffer_list, initial_buffer_list, policy, bufiter)

IN	name	name of object (MPIRT_STRING_NAME)
IN	maxbufcount	maximum number of buffers that can be in the buffer iterator (positive integer)
IN	bufsize	number of dataspec elements in a buffer (non-negative integer)
IN	dataspec	dataspec of the elements in a buffer (MPIRT_DATASPEC)
IN	allowed_buffer_list	container of all allowed buffers (MPIRT_CONTAINER)
IN	initial_buffer_list	*V-container* with initial buffers for the buffer iterator (MPIRT_CVECTOR)
IN	policy	buffer ordering policy (MPIRT_BUFITER_POLICY)
OUT	bufiter	buffer iterator (ref. to MPIRT_BUFITER)

```
int MPIRT_Bufiter_create (MPIRT_String_name name, int maxbufcount, int bufsize,
              MPIRT_Dataspec dataspec, MPIRT_Container allowed_buffer_list,
```

```
            MPIRT_Cvector initial_buffer_list, MPIRT_Bufiter_policy policy,
            MPIRT_Bufiter bufiter)

MPIRT::Bufiter (const MPIRT::String_Name name, int maxbufcount, int bufsize,
            const MPIRT::Dataspec &dataspec, const
            MPIRT::Container &allowed_buffer_list, const
            MPIRT::Cvector &initial_buffer_list, const
            MPIRT::Bufiter::Policy policy)
```

Rationale. The list of initial buffers is passed in as a *V-container* to properly insert initial buffers in the buffer iterator when MPIRT_BUFITER_FIFO, MPIRT_BUFITER_LIFO, or MPIRT_BUFITER_SORTED is used for policy. (*End of rationale.*)

An application is allowed to change both the parameters that define the iterator and which buffers should be placed into it at the start of the real-time mode until the buffer iterator is actually committed.

Four parameters are given to the buffer iterator to allow commit-time and run-time error checking and to allow an implementation to perform certain data transfer optimizations during commit-time. These parameters are a list of all allowed buffers to be transferred over the channel, called the allowed_buffer_list, the two parameters of dataspec and bufsize describing the data that will be transferred over the channel, and the policy of the buffer iterator. At buffer iterator create-time, any commit-time error that can be detected should result in the appropriate commit-time error being generated and the buffer iterator should not be created.

The allowed_buffer_list is specified as an MPIRT_CONTAINER and can be given as one of three types of values – the value MPIRT_CONTAINER_NULL, an *S-container*, or an *V-container*. The type of container given as this parameter implies certain information.

By giving the parameter as MPIRT_CONTAINER_NULL, the user is declaring that although the amount and type of data being transferred is known, as specified by the dataspec and bufsize parameters, the list of allowed buffers is not. This means that any buffer that matches the dataspec and bufsize values given to the buffer iterator is allowed to be inserted into the buffer iterator.

If the allowed_buffer_list is specified as an *S-container*, the allowed list of buffers is defined but it implies no order of production or consumption of these buffers by the user or the implementation. The *S-container* cannot be empty. All objects contained in the *S-container* must be of type MPIRT_BUFFER and cannot be MPIRT_OBJECT_NULL nor MPIRT_BUFFER_NULL. All buffers present in the *S-container* must be of the same dataspec and bufsize which must also be the same dataspec and bufsize that is given to the buffer iterator itself. Only buffers that are in the allowed_buffer_list should be inserted into the buffer iterator. This may allow an implementation to perform data transfer optimizations over a channel that uses this buffer iterator. At commit-time, if a buffer is present in the initial_buffer_list that is not in the allowed_buffer_list, the error MPIRT_ERR_ILLEGAL_BUFFER is generated.

If the allowed_buffer_list is specified as a *V-container*, the user implies more information to an implementation. The user states that the buffers will be inserted and consumed in the order that they are present in the allowed_buffer_list and the user indicates the order in which these buffers can be matched with remote buffers among the channel endpoints for data transfers (see Chapter 9). By specifying the allowed_buffer_list as a *V-container* in conjunction with certain buffer iterator policies defined below, the user makes a contract with an implementation that allows the implementation to perform even more aggressive optimizations for data transfers. This *V-container* cannot be empty. All objects contained in the *V-container* must be of type MPIRT_BUFFER and cannot be

MPIRT_OBJECT_NULL nor MPIRT_BUFFER_NULL. Only buffers that are in this *V-container* list are allowed to be inserted into the buffer iterator. All buffers present in the *V-container* must be of the same dataspec and bufsize which must also be the same dataspec and bufsize that is given to the buffer iterator itself. At commit-time, if a buffer is present in the initial_buffer_list that is not in the allowed_buffer_list, the error MPIRT_ERR_ILLEGAL_BUFFER is generated.

While a buffer object can be present in at most one buffer iterator at any given time and, hence, can only be in at most one initial_buffer_list of a buffer iterator at commit-time, the allowed_buffer_list does not have this restriction because many buffer iterators may eventually use a particular buffer object. Also because a buffer can be in at most one buffer iterator at a time, if an allowed_buffer_list is given, the maxbufcount parameter of the buffer iterator should be less than or equal to the size of the allowed_buffer_list. At commit-time, if any object in the container given for the allowed_buffer_list or the initial_buffer_list is not a buffer, the error MPIRT_ERR_NOT_A_BUFFER is generated.

The dataspec and bufsize parameters are used to describe the data that will be transferred over a channel which uses the buffer iterator. The dataspec must be a valid MPIRT_DATASPEC and cannot be MPIRT_DATASPEC_NULL. The bufsize parameter must be a non-negative integer. At commit-time, if any buffer in the allowed_buffer_list or the initial_buffer_list parameters whose dataspec and bufsize parameters do not match those of the buffer iterator, the error MPIRT_ERR_DATASPEC_MISMATCH is generated. Zero length buffers are allowed to be transferred over a channel, in which case, the dataspec given to the buffer iterator may be ignored by a implementation.

Since a buffer can be inserted into the buffer iterator either directly via an insert operation during run-time or indirectly by placing it into the buffer iterator initial_buffer_list before commit-time, error checking may be performed at both commit-time and run-time. At commit-time, the buffers present in the initial_buffer_list should be checked against the allowed_buffer_list and the buffers in both lists should be checked against the dataspec and bufsize of the buffer iterator. The buffers from the initial_buffer_list do not have to be in the buffer iterator until commit operation returns successfully, since insertion and removal operations on the buffer iterator object are not allowed until the object is committed. The MPIRT_COMMIT operation is responsible for inserting the buffers from the initial_buffer_list into the buffer iterator object if the MPIRT_COMMIT is successful.

For run-time error checking, any buffer inserted into the buffer iterator can be verified against the allowed_buffer_list or the dataspec and bufsize parameters during the MPIRT_BUFITER_INSERT operation. If the to-be-inserted buffer is not in the allowed_buffer_list or does not match the dataspec and bufsize of the buffer iterator, the run-time error MPIRT_ERR_ILLEGAL_BUFFER is returned by the MPIRT_BUFITER_INSERT operation and the buffer is not inserted into the buffer iterator.

The following four policies define how buffers are ordered upon insertion into the iterator. These four policies can be used with any allowed_buffer_list.

- MPIRT_BUFITER_FIFO specifies a first-in, first-out policy. That is, buffers are taken from the iterator in the order they were put into it. The order is not defined by labels.
- MPIRT_BUFITER_LIFO specifies a last-in, first-out policy. That is, buffers are taken from the iterator in the opposite order as they were put into it. The order is not defined by labels.
- MPIRT_BUFITER_SORTED specifies that buffers are ordered from lowest label to highest label. Since users define the labels, they can achieve any order they choose. For example, a priority scheme can be defined by assigning labels in reverse order.
- MPIRT_BUFITER_UNORDERED specifies that buffers are not ordered. That is, buffers may be taken out in any order.

If the allowed_buffer_list is given as a *V-container*, several other buffer iterator policies are allowed to be specified by the user to optimize data transfers further. MPIRT_BUFITER_FIFO_MAPPED indicates a mapping among all buffers residing in the same index value in all the allowed_buffer_lists in all the input buffer iterators for the channel endpoints. This requires that all of these allowed_buffer_lists contain the same number of buffers. The buffers are allowed to be inserted by the user in any order into the buffer iterator with the contract from the user that the mapped buffers will always be available for the transfer on all the channel endpoints otherwise, the program is erroneous. By using MPIRT_BUFITER_FIFO_ORDERED, the user promises that the buffers will be used cyclically in the order that they appear in the *V-container* beginning from index zero. This option gives an implementation the highest optimization potential as all transfers can be fully preplanned. MPIRT_BUFITER_LIFO_MAPPED and MPIRT_BUFITER_SORTED_MAPPED are both similar to their non-MAPPED counterparts except that they carry the contract from the user that there is a mapping among the buffers in the allowed_buffer_lists in the buffer iterators on the endpoints.

For all of the policies with mapping contracts, if the numbers of buffers at the endpoints are different, the commit-time error MPIRT_ERR_NOT_MAPPED is generated. Notice that there is no MPIRT_BUFITER_UNORDERED_MAPPED policy because the user cannot predict in which order that the buffers will be consumed by the channel and therefore, cannot guarantee that the correct buffer will be available on other endpoints for the transfer. If a policy specified by the user requires an *V-container* for the allowed_buffer_list parameter but an *V-container* is not given, the commit-time error MPIRT_ERR_INV_CVECTOR is generated.

Advice to implementors. For MPIRT_BUFITER_FIFO_ORDERED, an implementation may determine that mappings are possible among the buffers on the endpoints on an endpoint-by-endpoint basis based on the number of buffers present in the allowed_buffer_list of the buffer iterators of each endpoint. For point-to-point channels, these mappings may be one-to-one in the case of equal numbers, one-to-many in the case the numbers have common multiples, many-to-many, or all-to-all. An analogous analysis can be made for collective channels. (*End of advice to implementors.*)

$\perp$ (final)

Figure 5.8 illustrates the state transition diagram for buffer iterators. The state transition diagram depicts the two-phase process of buffer creation and the process by which buffers are moved in and out of the buffer iterator. As depicted in the figure, there are four conditions defined for a buffer iterator. The **overflow** (MPIRT_BUFITER_COND_OVERFLOW) occurs when the buffer iterator is full and a user or an implementation is trying to insert a buffer in it. The **underflow** (MPIRT_BUFITER_COND_UNDERFLOW) occurs when the buffer iterator is empty, and a user or an implementation is trying to remove a buffer from it. Both of these conditions are recoverable (see Chapter 4.2). The other two are MPIRT_BUFITER_COND_INSERTION and MPIRT_BUFITER_COND_REMOVAL, which occur upon successful insertion of a buffer to and removal of a buffer from an iterator, respectfully. These two conditions are not recoverable.

Advice to users. When the buffer iterator is full, and a user or an implementation tries to insert the buffer into it, the overflow condition is implicitly generated, and the implicit trigger is raised by the implementation. If there is no local explicit receptor or there are no synchronous handlers on the local explicit receptor for this condition, then the error MPIRT_ERR_BUFITER_OVERFLOW is returned from the insertion function. The state of the iterator is unchanged in this case – it acts as if the insertion function had not been called. If an event receptor is defined for this event, then a synchronous handler can be defined to correct this situation. A straightforward local synchronized handler will simply remove one buffer from the buffer iterator and then return MPIRT_SUCCESS. Recall that the only buffer that can be removed from the buffer iterator is the next buffer in the iterator as defined by

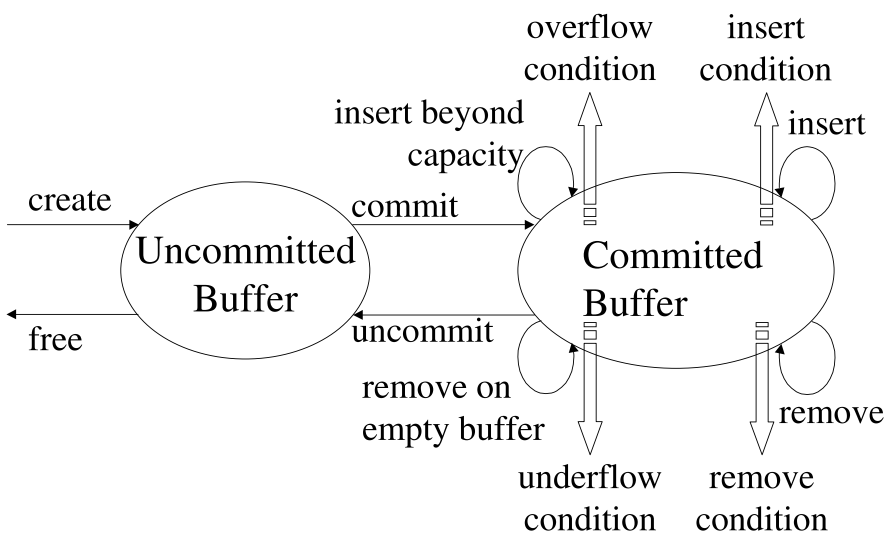

Figure 5.8: Buffer iterator state transition diagram.

the buffer iterator's buffer insertion policy. The user may then handle this removed buffer in some way relevant to the application (for example, throw it away or process it some other time).

Similarly, when the buffer iterator is empty and a user or an implementation tries to remove the buffer from it, the underflow condition is implicitly raised by the implementation. If there is no local explicit receptor or there are no synchronous handlers on the local explicit receptor for this condition, then the error MPIRT_ERR_BUFITER_UNDERFLOW is returned from the removal function. The state of the iterator is unchanged in this case – it acts as if the removal function had not been called. If the event receptor is defined for that event, then a handler for this event receptor is invoked analogously to the overflow condition. One way of handling this situation is to have a special buffer iterator with similarly-sized buffers of the same dataspec that is used only for the error handling. In case of error, the handler will just remove one buffer from this special buffer iterator, insert it into the buffer iterator which generated the underflow condition, and return MPIRT_SUCCESS to retry the operation. (*End of advice to users.*)

The two conditions for the buffer iterator objects defined above, namely MPIRT_BUFITER_COND_OVERFLOW and MPIRT_BUFITER_COND_UNDERFLOW, each raise an implicit event (see Chapter 4.2). An implementation creates an implicit trigger for each event. The event naming convention for these two events is the following: {user supplied bufiter name}:{rank of process that owns the buffer iterator}:{condition name}. The buffer iterator name is the object name of the buffer iterator object. The rank is the rank of ⊤ (final2)
the process where the buffer iterator resides with the group MPIRT_GROUP_WORLD.

For simplicity, the condition name used is shortened to "OVERFLOW" and "UNDERFLOW" respectively. For example, for a buffer iterator object named "input_iterator" created in a process ranked 2 in group MPIRT_GROUP_WORLD, the event name for buffer iterator overflow condition is ⊥ (final2) "input_iterator:2:OVERFLOW".

⊤ (final) The following operations query each individual parameter of a buffer iterator.

MPIRT_BUFITER_GET_MAXBUFCOUNT (bufiter, maxbufcount)

IN	bufiter	buffer iterator (MPIRT_BUFITER)
OUT	maxbufcount	maximum number of buffers that can be in the buffer iterator (ref. to positive integer)

```
int MPIRT_Bufiter_get_maxbufcount (MPIRT_Bufiter bufiter, int *maxbufcount)
```

```
int MPIRT::Bufiter::Get_Maxbufcount (int &maxbufcount) const
```

MPIRT_BUFITER_GET_DATASPEC (bufiter, dataspec)

IN	bufiter	buffer iterator (MPIRT_BUFITER)
OUT	dataspec	dataspec of the elements in a buffer (ref. to MPIRT_DATASPEC)

```
int MPIRT_Bufiter_get_dataspec (MPIRT_Bufiter bufiter, MPIRT_Dataspec *dataspec)
```

```
int MPIRT::Bufiter::Get_Dataspec (MPIRT::Dataspec *&dataspec) const
```

MPIRT_BUFITER_GET_BUFSIZE (bufiter, bufsize)

IN	bufiter	buffer iterator (MPIRT_BUFITER)
OUT	bufsize	number of dataspec elements in a buffer (ref. to a non-negative integer)

```
int MPIRT_Bufiter_get_bufsize (MPIRT_Bufiter bufiter, int *bufsize)
```

```
int MPIRT::Bufiter::Get_Bufsize (int &bufsize) const
```

MPIRT_BUFITER_GET_ALLOWED_BUFFER_LIST (bufiter, allowed_buffer_list)

IN	bufiter	buffer iterator (MPIRT_BUFITER)
OUT	allowed_buffer_list	a container of all allowed buffers (ref. to MPIRT_CONTAINER)

```
int MPIRT_Bufiter_get_allowed_buffer_list (MPIRT_Bufiter bufiter,
              MPIRT_Container *allowed_buffer_list)
```

```
int MPIRT::Bufiter::Get_Allowed_buffer_list
              (MPIRT::Container *&allowed_buffer_list) const
```

MPIRT_BUFITER_GET_INITIAL_BUFFER_LIST (bufiter, initial_buffer_list)

IN	bufiter	buffer iterator (MPIRT_BUFITER)
OUT	initial_buffer_list	a *V-container* with buffers (ref. to MPIRT_CVECTOR)

```
int MPIRT_Bufiter_get_initial_buffer_list (MPIRT_Bufiter bufiter,
              MPIRT_Cvector *initial_buffer_list)
```

```
int MPIRT::Bufiter::Get_Initial_buffer_list (MPIRT::Cvector *&initial_buffer_list)
              const
```

MPIRT_BUFITER_GET_POLICY (bufiter, policy)

IN	bufiter	buffer iterator (MPIRT_BUFITER)
OUT	policy	buffer-ordering policy (ref. to MPIRT_BUFITER_POLICY)

```
int MPIRT_Bufiter_get_policy (MPIRT_Bufiter bufiter, MPIRT_Bufiter_policy *policy)
```

```
int MPIRT::Bufiter::Get_Policy (MPIRT::Bufiter::Policy &policy) const
```

The following operations set each individual parameter of a buffer iterator.

MPIRT_BUFITER_SET_MAXBUFCOUNT (bufiter, maxbufcount)

INOUT	bufiter	buffer iterator (ref. to MPIRT_BUFITER)
IN	maxbufcount	maximum number of buffers that the buffer iterator can have (positive integer)

```
int MPIRT_Bufiter_set_maxbufcount (MPIRT_Bufiter *bufiter, int maxbufcount)
```

```
int MPIRT::Bufiter::Set_Maxbufcount (int maxbufcount)
```

MPIRT_BUFITER_SET_DATASPEC (bufiter, dataspec)

INOUT	bufiter	buffer iterator (MPIRT_BUFITER)
IN	dataspec	dataspec of the elements in a buffer (MPIRT_DATASPEC)

```
int MPIRT_Bufiter_set_dataspec (MPIRT_Bufiter *bufiter, MPIRT_Dataspec dataspec)
```

```
int MPIRT::Bufiter::Set_Dataspec (const MPIRT::Dataspec &dataspec)
```

MPIRT_BUFITER_SET_BUFSIZE (bufiter, bufsize)

INOUT	bufiter	buffer iterator (MPIRT_BUFITER)
IN	bufsize	number of dataspec elements in a buffer (a non-negative integer)

```
int MPIRT_Bufiter_set_bufsize (MPIRT_Bufiter *bufiter, int bufsize)
```

```
int MPIRT::Bufiter::Set_Bufsize (int bufsize)
```

MPIRT_BUFITER_SET_ALLOWED_BUFFER_LIST (bufiter, initial_buffer_list)

INOUT	bufiter	buffer iterator (MPIRT_BUFITER)
IN	allowed_buffer_list	a container of all allowed buffers (MPIRT_CONTAINER)

```
int MPIRT_Bufiter_set_allowed_buffer_list (MPIRT_Bufiter *bufiter,
              MPIRT_Container allowed_buffer_list)
```

```
int MPIRT::Bufiter::Set_Allowed_buffer_list (const
              MPIRT::Container &allowed_buffer_list)
```

MPIRT_BUFITER_SET_INITIAL_BUFFER_LIST (bufiter, initial_buffer_list)

INOUT	bufiter	buffer iterator (ref. to MPIRT_BUFITER)
IN	initial_buffer_list	a *V-container* with initial buffers for the buffer iterator (MPIRT_CVECTOR)

```
int MPIRT_Bufiter_set_initial_buffer_list (MPIRT_Bufiter *bufiter,
              MPIRT_Cvector initial_buffer_list)
```

```
int MPIRT::Bufiter::Set_Initial_buffer_list (const
              MPIRT::Cvector &initial_buffer_list)
```

MPIRT_BUFITER_SET_POLICY (bufiter, policy)

INOUT	bufiter	buffer iterator (ref. to MPIRT_BUFITER)
IN	policy	buffer ordering policy (MPIRT_BUFITER_POLICY)

```
int MPIRT_Bufiter_set_policy (MPIRT_Bufiter *bufiter, MPIRT_Bufiter_policy policy)
```

```
int MPIRT::Bufiter::Set_Policy (const MPIRT::Bufiter::Policy &policy) const
```

⊥ (final)

The duplication, free, and comparison operations and the set and get name parameter operations for buffer iterators are inherited from the general class of MPIRT_OBJECT (see Section 2.2).

5.5 Buffer Iterator Accessors

These operations allow users to put buffers into or take buffers from a buffer iterator. These operations are only defined for the committed buffer iterator. If a user invokes these operation on an uncommitted buffer iterator MPI/RT will return MPIRT_ERR_OBJECT_NOT_COMMITTED error. Any buffer put into an iterator must match bufsize and dataspec of the buffers in initial_buffer_list and be from allowed_buffer_list. No buffers can be inserted or removed until the iterator is in the committed state. Although the buffer iterator accessor functions themselves may be thread-safe, users are responsible for synchronization between multiple threads which may simultaneously access a buffer iterator by use of QoS scheduling or by other methods.

An attempt to insert a buffer into a full buffer iterator generates a recoverable condition called MPIRT_BUFITER_COND_OVERFLOW. If a local explicit receptor for this buffer iterator's overflow condition exists, this operation does not return until a synchronous handler attached to the receptor returns MPIRT_SUCCESS or MPIRT_FAILURE, or all of them return MPIRT_CONTINUE. If MPIRT_SUCCESS is returned from the handlers, the implementation will retry the insertion operation, which may cause this condition to be generated again. If the insertion is then successful, the return value from this function is MPIRT_SUCCESS. If MPIRT_FAILURE is returned from a handler or MPIRT_CONTINUE is returned by all the synchronous handlers, the insertion operation is not retried, and the insertion has failed. MPIRT_ERR_BUFITER_OVERFLOW is returned from this function, and the buffer has not been inserted.

The MPIRT_BUFITER_INSERT operation puts a buffer into a buffer iterator, according to the ordering policy defined by iterator policy.

MPIRT_BUFITER_INSERT (bufiter, buffer)

INOUT	bufiter	buffer iterator (ref. to MPIRT_BUFITER)
IN	buffer	buffer (MPIRT_BUFFER)

```
int MPIRT_Bufiter_insert (MPIRT_Bufiter *bufiter, MPIRT_Buffer buffer)
```

```
int MPIRT::Bufiter::Insert (const MPIRT::Buffer &buffer)
```

The MPIRT_BUFITER_REMOVE operation removes the next buffer from the buffer iterator as specified by the ordering policy when the buffer was inserted.

An attempt to remove a buffer from an empty buffer iterator generates a recoverable condition called MPIRT_BUFITER_COND_UNDERFLOW. If a local explicit receptor for this buffer iterator's underflow condition exists, this operation does not return until a synchronous handler attached to the receptor returns MPIRT_SUCCESS or MPIRT_FAILURE, or all of them return MPIRT_CONTINUE. If MPIRT_SUCCESS is returned from the handlers, the implementation will retry the removal operation, which may cause this condition to be generated again. If the removal is then successful, the return value from this function is MPIRT_SUCCESS. If MPIRT_FAILURE is returned from a handler, or MPIRT_CONTINUE is returned by all the synchronous handlers, the removal operation is not retried, and the removal has failed. MPIRT_ERR_BUFITER_UNDERFLOW is returned from this function, and no buffer has been removed from the iterator.

MPIRT_BUFITER_REMOVE (bufiter, buffer)

INOUT	bufiter	buffer iterator (ref. to MPIRT_BUFITER)
OUT	buffer	buffer (MPIRT_BUFFER)

```
int MPIRT_Bufiter_remove (MPIRT_Bufiter *bufiter, MPIRT_Buffer *buffer)
```

```
int MPIRT::Bufiter::Remove (MPIRT::Buffer *&buffer)
```

5.6 Summary

This section summarizes enumerations and type declarations that are introduced in this chapter.

Micellaneous Definitions		
Item	*What is it*	*Defined as:*
MPIRT_BUFFER	buffer object	buffer object
MPIRT_BUFITER	buffer iterator object	buffer iterator object
MPIRT_OBJECT_TYPE_BUFFER	a buffer's MPIRT_OBJECT_TYPE	enumeration
MPIRT_OBJECT_TYPE_BUFITER	a buffer iterator's MPIRT_OBJECT_TYPE	enumeration
MPIRT_BUFFER_NULL	null buffer	null buffer object
MPIRT_BUFITER_NULL	null buffer iterator	null buffer iterator object
MPIRT_BUFFER_LABEL_DEFAULT	default label for a buffer	enumeration
MPIRT_SYSTEM_MEM_ALLOC	specifies the system should allocate memory for the buffer	enumeration

⊤ (final2)
⊥ (final2)
⊤ (final2)
⊥ (final2)

Buffer Iterator Policies	
Policy	*Definition*
MPIRT_BUFITER_FIFO	first-in, first-out (queue)
MPIRT_BUFITER_LIFO	last-in, first-out (stack)
MPIRT_BUFITER_SORTED	sorted based on buffer label (priority queue)
MPIRT_BUFITER_UNORDERED	implementation's choice
MPIRT_BUFITER_FIFO_MAPPED	first-in, first-out (queue) with allowed one-to-one mapping between buffers in channel endpoints buffer iterators
MPIRT_BUFITER_FIFO_ORDERED	first-in, first-out (queue) with a guarantee that the buffers in allowed_buffer_list will always appear cyclically in the buffer iterator, and allowed one-to-one mapping between buffers in channel endpoints buffer iterators
MPIRT_BUFITER_LIFO_MAPPED	last-in, first-out (stack) with allowed one-to-one mapping between buffers in channel endpoints buffer iterators
MPIRT_BUFITER_SORTED_MAPPED	sorted based on buffer label (priority queue) with allowed one-to-one mapping between buffers in channel endpoints buffer iterators

Buffer Iterator Conditions			
Condition Enumeration	*When Raised*	*Name String*	*Recoverable*
MPIRT_BUFITER_COND_OVERFLOW	iterator overflow	"OVERFLOW"	yes
MPIRT_BUFITER_COND_UNDERFLOW	iterator underflow	"UNDERFLOW"	yes
MPIRT_BUFITER_COND_INSERTION	buffer inserted into iterator	"INSERTION"	no
MPIRT_BUFITER_COND_REMOVAL	buffer removed from iterator	"REMOVAL"	no

Buffer Specific Operations		
Language Independent Specification	C	C++
MPIRT_BUFFER_CREATE	MPIRT_Buffer_create	MPIRT::Buffer
MPIRT_BUFFER_GET_BUFSIZE	MPIRT_Buffer_get_bufsize	MPIRT::Buffer::Get_Bufsize
MPIRT_BUFFER_GET_DATASPEC	MPIRT_Buffer_get_dataspec	MPIRT::Buffer::Get_Dataspec
MPIRT_BUFFER_GET_BASE	MPIRT_Buffer_get_base	MPIRT::Buffer::Get_Base
MPIRT_BUFFER_SET_BUFSIZE	MPIRT_Buffer_set_bufsize	MPIRT::Buffer::Set_Bufsize
MPIRT_BUFFER_SET_DATASPEC	MPIRT_Buffer_set_dataspec	MPIRT::Buffer::Set_Dataspec
MPIRT_BUFFER_SET_BASE	MPIRT_Buffer_set_base	MPIRT::Buffer::Set_Base
MPIRT_BUFFER_ASSIGN_LABEL	MPIRT_Buffer_assign_label	MPIRT::Buffer::Assign_Label
MPIRT_BUFFER_RETRIEVE_LABEL	MPIRT_Buffer_retrieve_label	MPIRT::Buffer::Retrieve_Label
MPIRT_BUFFER_GET_NAME	MPIRT_Buffer_get_name	MPIRT::Buffer::Get_Name
MPIRT_BUFFER_SET_NAME	MPIRT_Buffer_set_name	MPIRT::Buffer::Set_Name
MPIRT_BUFFER_DUP	MPIRT_Buffer_dup	MPIRT::Buffer MPIRT::Buffer::Dup MPIRT::Buffer::Operator=
MPIRT_BUFFER_FREE	MPIRT_Buffer_free	MPIRT::~Buffer MPIRT::Buffer::Free
MPIRT_BUFFER_IS_EQUAL	MPIRT_Buffer_is_equal	MPIRT::Buffer::Is_equal
MPIRT_BUFFER_ATTACH_ATTR	MPIRT_Buffer_attach_attr	MPIRT::Buffer::Attach_Attr
MPIRT_BUFFER_RETRIEVE_ATTR	MPIRT_Buffer_retrieve_attr	MPIRT::Buffer::Retrieve_Attr
MPIRT_BUFFER_DELETE_ATTR	MPIRT_Buffer_delete_attr	MPIRT::Buffer::Delete_Attr
MPIRT_BUFFER_CREATE_SUBBUFFERS	MPIRT_Buffer_create_buffers	MPIRT::Cvector
MPIRT_BUFFER_CREATE_SUBBUFFER	MPIRT_Buffer_create_subset	MPIRT::Buffer

Buffer Iterator Specific Operations		
Language Independent Specification	C	C++
MPIRT_BUFITER_CREATE	MPIRT_Bufiter_create	MPIRT::Bufiter
MPIRT_BUFITER_GET_MAXBUFCOUNT	MPIRT_Bufiter_get_maxbufcount	MPIRT::Bufiter::Get_Maxbufcount
MPIRT_BUFITER_GET_DATASPEC	MPIRT_Bufiter_get_dataspec	MPIRT::Bufiter::Get_Dataspec
MPIRT_BUFITER_GET_BUFSIZE	MPIRT_Bufiter_get_maxbufcount	MPIRT::Bufiter::Get_Maxbufcount
MPIRT_BUFITER_GET_ALLOWED_BUFFER_LIST	MPIRT_Bufiter_get_allowed_buffer_list	MPIRT::Bufiter::Get_Allowed_buffer_list
MPIRT_BUFITER_GET_INITIAL_BUFFER_LIST	MPIRT_Bufiter_get_initial_buffer_list	MPIRT::Bufiter::Get_Initial_buffer_list
MPIRT_BUFITER_GET_POLICY	MPIRT_Bufiter_get_policy	MPIRT::Bufiter::Get_Policy
MPIRT_BUFITER_SET_MAXBUFCOUNT	MPIRT_Bufiter_set_maxbufcount	MPIRT::Bufiter::Set_Maxbufcount
MPIRT_BUFITER_SET_DATASPEC	MPIRT_Bufiter_set_dataspec	MPIRT::Bufiter::Set_Dataspec
MPIRT_BUFITER_SET_BUFSIZE	MPIRT_Bufiter_set_maxbufcount	MPIRT::Bufiter::Set_Maxbufcount
MPIRT_BUFITER_SET_ALLOWED_BUFFER_LIST	MPIRT_Bufiter_set_allowed_buffer_list	MPIRT::Bufiter::Set_Allowed_buffer_list
MPIRT_BUFITER_SET_INITIAL_BUFFER_LIST	MPIRT_Bufiter_set_initial_buffer_list	MPIRT::Bufiter::Set_initial_buffer_list
MPIRT_BUFITER_SET_POLICY	MPIRT_Bufiter_set_policy	MPIRT::Bufiter::Set_Policy
MPIRT_BUFITER_GET_NAME	MPIRT_Bufiter_get_name	MPIRT::Bufiter::Get_Name
MPIRT_BUFITER_SET_NAME	MPIRT_Bufiter_set_name	MPIRT::Bufiter::Set_Name
MPIRT_BUFITER_DUP	MPIRT_Bufiter_dup	MPIRT::Bufiter MPIRT::Bufiter::Dup MPIRT::Bufiter::Operator=
MPIRT_BUFITER_FREE	MPIRT_Bufiter_free	MPIRT::∼Bufiter MPIRT::Bufiter::Free
MPIRT_BUFITER_IS_EQUAL	MPIRT_Bufiter_is_equal	MPIRT::Bufiter::Is_equal
MPIRT_BUFITER_ATTACH_ATTR	MPIRT_Bufiter_attach_attr	MPIRT::Bufiter::Attach_Attr
MPIRT_BUFITER_RETRIEVE_ATTR	MPIRT_Bufiter_retrieve_attr	MPIRT::Bufiter::Retrieve_Attr
MPIRT_BUFITER_DELETE_ATTR	MPIRT_Bufiter_delete_attr	MPIRT::Bufiter::Delete_Attr
MPIRT_BUFITER_INSERT	MPIRT_Bufiter_insert	MPIRT::Bufiter::Insert
MPIRT_BUFITER_REMOVE	MPIRT_Bufiter_remove	MPIRT::Bufiter::Remove

Micellaneous Bindings		
Item	*C Binding*	*C++ Binding*
MPIRT_BUFFER	MPIRT_Buffer	MPIRT::Buffer
MPIRT_BUFITER	MPIRT_Bufiter	MPIRT::Bufiter
MPIRT_OBJECT_TYPE_BUFFER	MPIRT_OBJECT_TYPE_BUFFER	MPIRT::OBJECT_TYPE_BUFFER
MPIRT_OBJECT_TYPE_BUFITER	MPIRT_OBJECT_TYPE_BUFITER	MPIRT::OBJECT_TYPE_BUFITER
MPIRT_BUFFER_NULL	MPIRT_BUFFER_NULL	MPIRT::BUFFER_NULL
MPIRT_BUFITER_NULL	MPIRT_BUFITER_NULL	MPIRT::BUFITER_NULL
MPIRT_BUFFER_LABEL_DEFAULT	MPIRT_BUFFER_LABEL_DEFAULT	MPIRT::Buffer::LABEL_DEFAULT
MPIRT_SYSTEM_MEM_ALLOC	MPIRT::SYSTEM_MEM_ALLOC	MPIRT::SYSTEM_MEM_ALLOC
MPIRT_BUFITER_FIFO	MPIRT_BUFITER_FIFO	MPIRT::Bufiter::FIFO
MPIRT_BUFITER_LIFO	MPIRT_BUFITER_LIFO	MPIRT::Bufiter::LIFO
MPIRT_BUFITER_UNORDERED	MPIRT_BUFITER_UNORDERED	MPIRT::Bufiter::UNORDERED
MPIRT_BUFITER_FIFO_MAPPED	MPIRT_BUFITER_FIFO_MAPPED	MPIRT::Bufiter::FIFO_MAPPED
MPIRT_BUFITER_FIFO_ORDERED	MPIRT_BUFITER_FIFO_ORDERED	MPIRT::Bufiter::FIFO_ORDERED
MPIRT_BUFITER_LIFO_MAPPED	MPIRT_BUFITER_LIFO_MAPPED	MPIRT::Bufiter::LIFO_MAPPED
MPIRT_BUFITER_SORTED_MAPPED	MPIRT_BUFITER_SORTED_MAPPED	MPIRT::Bufiter::SORTED_MAPPED
MPIRT_BUFITER_COND_OVERFLOW	MPIRT_BUFITER_COND_OVERFLOW	MPIRT::Bufiter::COND_OVERFLOW
MPIRT_BUFITER_COND_UNDERFLOW	MPIRT_BUFITER_COND_UNDERFLOW	MPIRT::Bufiter::COND_UNDERFLOW
MPIRT_BUFITER_COND_INSERTION	MPIRT_BUFITER_COND_INSERTION	MPIRT::Bufiter::COND_INSERTION
MPIRT_BUFITER_COND_REMOVAL	MPIRT_BUFITER_COND_REMOVAL	MPIRT::Bufiter::COND_REMOVAL
MPIRT_ERR_NOT_A_BUFFER	MPIRT_ERR_NOT_A_BUFFER	MPIRT::ERR_NOT_A_BUFFER
MPIRT_ERR_ILLEGAL_BUFFER	MPIRT_ERR_ILLEGAL_BUFFER	MPIRT::ERR_ILLEGAL_BUFFER
MPIRT_ERR_BUFITER_OVERFLOW	MPIRT_ERR_BUFITER_OVERFLOW	MPIRT::ERR_BUFITER_OVERFLOW
MPIRT_ERR_BUFITER_UNDERFLOW	MPIRT_ERR_BUFITER_UNDERFLOW	MPIRT::ERR_BUFITER_UNDERFLOW

Part II

Transfer Mechanisms and Advanced Objects

Chapter 6

Channel Overview

Current Status: First formal vote passed 8/0/0 in May 98.
Second formal vote passed 8/0/3 in July 98. Third formal vote passed 9/0/0 on 12/3/98.

6.1 Introduction

Channels in MPI/RT offer the functionality of a "persistent virtual connection" or virtual channel [4, 6, 13]. The motivations for having virtual channels in MPI/RT are threefold:

1. More efficient use of the resources by the implementation including the ability to exploit persistent communication (planned transfer),
2. Deadlock avoidance, and
3. The ability to provide quality-of-service (QoS) guarantees and timing correctness.

The object directly associated with the message transfers is a channel that contains buffer iterators and a QoS specification. MPIRT_CHANNEL_START starts the data transfer but does not wait for the completion[1]. The QoS information of the channel (point-to-point or collective) provides the user with control over the data transfer operation, and events for the channel data transfer with event receptor handlers for the channel conditions are capable of providing applications with status (*e.g.,* completion) information.

MPIRT_CHANNEL_ACTIVATE asserts that the user will not be issuing any explicit data transfer operations (MPIRT_CHANNEL_START), but instead the data transfers will be controlled by the QoS parameters of the MPIRT_CHANNEL: point-to-point channel or collective channel (an implementation will be transferring the data according to the time schedule or events), or by the other side of a point-to-point channel, or by some other side of a collective channel. For the time-driven paradigm, the commit operation generates the time schedule using on-line or off-line facilities or by loading an already built one. In that case, MPIRT_CHANNEL_ACTIVATE loads its calendar and follows its schedule. MPIRT_CHANNEL_DEACTIVATE turns it off. For the event-driven paradigm, the data transfer starts and stops based on events defined by the QoS_constraints of the QoS of the channel (see Chapter 10). ⊤ (final)

A point-to-point channel references an input buffer iterator and an output buffer iterator. The allowed_buffer_list given to the input buffer iterator must be equivalent to the allowed_buffer_list of the

[1]All data transfer operations inherently use the **ready** mode concepts of MPI even though they are not restricted to the *push* only model of the MPI-1.

output buffer iterator in the following sense. Both lists must be of the same container type and the buffers referenced by the lists must be the same. If the list is given as MPIRT_CONTAINER_NULL, both buffer iterators must give the list as MPIRT_CONTAINER_NULL. If the list is given as *S-container*, both buffer iterators must supply *S-containers* that contain the same buffer objects but not necessarily in the same order. If the list is given as *V-container*, both buffer iterators must supply *V-containers* that contain the same buffers in the same order given by the same indices. This can be achieved by giving the same container as the allowed_buffer_list to both buffer iterators. At commit-time, the error MPIRT_ERR_ILLEGAL_BUFITER will be generated if these qualifications are not met. Many collective channels have two pairs of input and output buffer iterators to perform their function. These requirements apply to each pair of buffer iterators for those collective channels.

The specification of buffer iterator policies and lists of allowed buffers allow implementations $\perp$ (final) to optimize data transfers (see Chapters 5.4 and 9).

The completion of a message transfer can be ascertained by using MPIRT_CHANNEL_WAIT, while MPIRT_CHANNEL_TEST can be used to determine the state of a channel.

QoS in the context of a virtual channel is established prior to the use of that channel and includes control of underlying resources, including, but not limited to, end-to-end delay control, jitter control, bandwidth control, and system buffer space control.

The point-to-point channels and collective channels are subclasses of MPIRT_CHANNEL. The objects of type MPIRT_CHANNEL can be grouped together into a container so that they can all be committed at the same time.

The creation of point-to-point channels is defined in Chapter 7, and the creation of collective channels is defined in Chapter 8.

The name is the name that is used by the implementation to match sides of the channel. Thus the name should be unique over the scope of the group over which the channel is defined. The same $\top$ (final2) name is used for the event name for the channel endpoint events.

Event names for each channel condition are named using the following system: {*channel name*}:{*rank*}:{*channel event*}. This naming convention uniquely identifies an event within the scope of the group over which the channel is defined. The rank represents the rank of the channel endpoint where the condition is raised. For example, if a user is interested in the completion of a data transfer on the sender side of a point-to-point channel named *channel#1* between the sender rank 1 and the receiver rank 3 over the group *my_group*, the application can create a receptor in any process with the event name channel#1:1:COMP.

Since the event name should be unique for creating a match between triggers and receptors, the object name must be unique with the process defined by the rank if there is a receptor for the event name. For channel objects that also means that the channel name is unique not only over the scope of the group over which the channel is defined but also over the scope of group MPIRT_GROUP_WORLD. This means that the channel name is unique with respect to all channels if there exist a receptor for the channel condition. For the channels that do not have a local or remote, implicit or explicit receptor defined for a channel condition the object name for the channel $\perp$ (final2) object is not required to be unique.

$\top$ (final) These channel events are as follows:

- MPIRT_CHANNEL_COND_COMP,
- MPIRT_CHANNEL_COND_START, and
- MPIRT_CHANNEL_COND_QOS_ERROR

$\top$ (final2)

Users can create explicit event receptors for these events or use these event names for a guard $\perp$ (final2) for some channel. These event receptors and channels must be part of the same MPIRT_COMMIT.

The QoS of the channel is of the type MPIRT_QOS_CHANNEL, as defined in Section 11.2. The QoS specification can be one of the following:

1. MPIRT_QOS_CHANNEL_TIME for the time-driven paradigm,
2. MPIRT_QOS_CHANNEL_PREV for the event-driven with priority paradigm,
3. MPIRT_QOS_CHANNEL_PRTIEV for the mixture of event-time-driven with priority paradigm,
4. the constant MPIRT_QOS_ACCEPT_ANY, which allows single-ended specification of a channel's QoS requirements, or
5. the constant MPIRT_QOS_BEST_EFFORT for the best QoS MPI/RT can provide without violating other objects' QoS guarantees.

Normally, the QoS specification for both ends of a point-to-point channel specification must be identical, except when one end specifies MPIRT_QOS_ACCEPT_ANY. If both endpoints specify MPIRT_QOS_ACCEPT_ANY, it is equivalent to specifying MPIRT_QOS_BEST_EFFORT.

If the QoS for all channel endpoints is of type MPIRT_QOS_CHANNEL_PREV or MPIRT_QOS_CHANNEL_PRTIEV then the single occurrence of the event that is specified in the QoS that starts the message transfer transfers only one message over the channel.

Each point-to-point channel endpoint has two iterators: one for input, inbufiter, from which buffers are taken to be sent or received, and another for output, outbufiter, into which buffers are put after a channel operation completes. Collective channel endpoints may have more than one pair of in and out buffer iterators whenever an endpoint is involved in both sending and receiving of messages.

6.2 Common Attributes Operations for Channels

This section describes operations on attributes common to all channel types. These include inbufiter and outbufiter, group, and channel_qos. Since name is common for all MPI/RT objects, the query and set operations for it are defined on the base object, MPIRT_OBJECT, and appear in Section 2.2. The other common operations on the channels – namely dup, free, and compare – are also defined in Section 2.2.

The following operations query each individual parameter of the channel object.

MPIRT_CHANNEL_GET_INBUFITER (channel, inbufiter)

IN	channel	channel (MPIRT_CHANNEL)
OUT	inbufiter	input buffer iterator of the channel (ref. to MPIRT_BUFITER)

```
int MPIRT_Channel_get_inbufiter (MPIRT_Channel channel, MPIRT_Bufiter *inbufiter)
```

```
int MPIRT::Channel::Get_Inbufiter (MPIRT::Bufiter *&inbufiter) const
```

MPIRT_CHANNEL_GET_OUTBUFITER (channel, outbufiter)

IN	channel	channel (MPIRT_CHANNEL)
OUT	outbufiter	output buffer iterator for the channel (ref. to MPIRT_BUFITER)

```
int MPIRT_Channel_get_outbufiter (MPIRT_Channel channel, MPIRT_Bufiter *outbufiter)
```

```
int MPIRT::Channel::Get_Outbufiter (MPIRT::Bufiter *&outbufiter) const
```

MPIRT_CHANNEL_GET_GROUP (channel, group)

IN	channel	channel (MPIRT_CHANNEL)
OUT	group	group over which the channel operates (ref. to MPIRT_GROUP)

```
int MPIRT_Channel_get_group (MPIRT_Channel channel, MPIRT_Group *group)
```

```
int MPIRT::Channel::Get_Group (MPIRT::Group *&group) const
```

MPIRT_CHANNEL_GET_QOS (channel, channel_qos)

IN	channel	channel (MPIRT_CHANNEL)
OUT	channel_qos	QoS for the channel (ref. to MPIRT_QOS_CHANNEL)

```
int MPIRT_Channel_get_qos (MPIRT_Channel channel, MPIRT_Qos_channel *channel_qos)
```

```
int MPIRT::Channel::Get_Qos (MPIRT::Qos_Channel *&channel_qos) const
```

The following operations allow a user to set each individual parameter of the channel object.

MPIRT_CHANNEL_SET_INBUFITER (channel, inbufiter)

INOUT	channel	channel (ref. to MPIRT_CHANNEL)
IN	inbufiter	input buffer iterator of the channel (MPIRT_BUFITER)

```
int MPIRT_Channel_set_inbufiter (MPIRT_Channel *channel, MPIRT_Bufiter inbufiter)
```

```
int MPIRT::Channel::Set_Inbufiter (const MPIRT::Bufiter &inbufiter)
```

MPIRT_CHANNEL_SET_OUTBUFITER (channel, outbufiter)

INOUT	channel	channel (ref. to MPIRT_CHANNEL)
IN	outbufiter	output buffer iterator of the channel (MPIRT_BUFITER)

```
int MPIRT_Channel_set_outbufiter (MPIRT_Channel *channel, MPIRT_Bufiter outbufiter)
```

```
int MPIRT::Channel::Set_Outbufiter (const MPIRT::Bufiter &outbufiter)
```

MPIRT_CHANNEL_SET_GROUP (channel, group)

INOUT	channel	channel (ref. to MPIRT_CHANNEL)
IN	group	group over which the channel operates (MPIRT_GROUP)

```
int MPIRT_Channel_set_group (MPIRT_Channel *channel, MPIRT_Group group)
```

```
int MPIRT::Channel::Set_Group (const MPIRT::Group &group)
```

MPIRT_CHANNEL_SET_QOS (channel, channel_qos)

INOUT	channel	channel (ref. to MPIRT_CHANNEL)
IN	channel_qos	QoS for the channel (MPIRT_QOS_CHANNEL)

```
int MPIRT_Channel_set_qos (MPIRT_Channel *channel, MPIRT_Qos_channel channel_qos)
```

```
int MPIRT::Channel::Set_Qos (const MPIRT::Qos_Channel &channel_qos)
```

⊥ (final2)

6.3 Summary

This section summarizes enumerations and type declarations that are introduced in this chapter.

Miscellaneous Channel Definitions

Item	*What is it*	*Defined as:*
MPIRT_CHANNEL_NULL	null channel	null channel object
MPIRT_QOS_ACCEPT_ANY	parameter value to accept other channel endpoint's(s') quality-of-service	MPIRT_QOS
MPIRT_QOS_BEST_EFFORT	MPI/RT best available non guarantee QoS	MPIRT_QOS

Channel Conditions

Condition Enumeration	*When Raised*	*Name String*	*Recoverable*
MPIRT_CHANNEL_COND_COMP	completion of a channel transfer	"COMPLETION"	no
MPIRT_CHANNEL_COND_START	start of a channel transfer	"START"	no
MPIRT_CHANNEL_COND_QOS_ERROR	quality-of-service error on channel	"QOS_ERROR"	no

⊤ (final2)
⊥ (final2)
⊥ (final)

General Channel Operations		
Language Independent Specification	C	C++
MPIRT_CHANNEL_GET_INBUFITER	MPIRT_Channel_get_inbufiter	MPIRT::Channel::Get_Inbufiter
MPIRT_CHANNEL_GET_OUTBUFITER	MPIRT_Channel_get_outbufiter	MPIRT::Channel::Get_Outbufiter
MPIRT_CHANNEL_GET_GROUP	MPIRT_Channel_get_group	MPIRT::Channel::Get_Group
MPIRT_CHANNEL_GET_QOS	MPIRT_Channel_get_qos	MPIRT::Channel::Get_Qos
MPIRT_CHANNEL_SET_INBUFITER	MPIRT_Channel_set_inbufiter	MPIRT::Channel::Set_Inbufiter
MPIRT_CHANNEL_SET_OUTBUFITER	MPIRT_Channel_set_outbufiter	MPIRT::Channel::Set_Outbufiter
MPIRT_CHANNEL_SET_GROUP	MPIRT_Channel_set_group	MPIRT::Channel::Set_Group
MPIRT_CHANNEL_SET_QOS	MPIRT_Channel_set_qos	MPIRT::Channel::Set_Qos
MPIRT_CHANNEL_START	MPIRT_Channel_start	MPIRT::Channel::Start
MPIRT_CHANNEL_ACTIVATE	MPIRT_Channel_activate	MPIRT::Channel::Activate
MPIRT_CHANNEL_DEACTIVATE	MPIRT_Channel_deactivate	MPIRT::Channel::Deactivate
MPIRT_CHANNEL_WAIT	MPIRT_Channel_wait	MPIRT::Channel::Wait
MPIRT_CHANNEL_TEST	MPIRT_Channel_test	MPIRT::Channel::Test
MPIRT_CHANNEL_GET_NAME	MPIRT_Channel_get_name	MPIRT::Channel::Get_Name
MPIRT_CHANNEL_SET_NAME	MPIRT_Channel_set_name	MPIRT::Channel::Set_Name
MPIRT_CHANNEL_DUP	MPIRT_Channel_dup	MPIRT::Channel MPIRT::Channel::Dup MPIRT::Channel::Operator=
MPIRT_CHANNEL_FREE	MPIRT_Channel_free	MPIRT::∼Channel MPIRT::Channel::Free
MPIRT_CHANNEL_IS_EQUAL	MPIRT_Channel_is_equal	MPIRT::Channel::Is_equal
MPIRT_CHANNEL_ATTACH_ATTR	MPIRT_Channel_attach_attr	MPIRT::Channel::Attach_Attr
MPIRT_CHANNEL_RETRIEVE_ATTR	MPIRT_Channel_retrieve_attr	MPIRT::Channel::Retrieve_Attr
MPIRT_CHANNEL_DELETE_ATTR	MPIRT_Channel_delete_attr	MPIRT::Channel::Delete_Attr

Miscellaneous Bindings		
Item	*C Bindings*	*C++ Bindings*
MPIRT_CHANNEL	MPIRT_Channel	MPIRT::Channel
MPIRT_CHANNEL_NULL	MPIRT_CHANNEL_NULL	MPIRT::CHANNEL_NULL
MPIRT_QOS_ACCEPT_ANY	MPIRT_QOS_ACCEPT_ANY	MPIRT::QOS_ACCEPT_ANY
MPIRT_QOS_BEST_EFFORT	MPIRT_QOS_BEST_EFFORT	MPIRT::QOS_BEST_EFFORT
MPIRT_CHANNEL_COND_COMP	MPIRT_CHANNEL_COND_COMP	MPIRT::Channel::COND_COMP
MPIRT_CHANNEL_COND_START	MPIRT_CHANNEL_COND_START	MPIRT::Channel::COND_START
MPIRT_CHANNEL_COND_QOS_ERROR	MPIRT_CHANNEL_COND_QOS_ERROR	MPIRT::Channel::COND_QOS_ERROR
MPIRT_ERR_ILLEGAL_BUFITER	MPIRT_ERR_ILLEGAL_BUFITER	MPIRT::ERR_ILLEGAL_BUFITER

Chapter 7

Point-to-Point Channels

Current Status: First reading vote passed 7/0/0, 5/20/98.
Second vote passed 9/0/2, 7/98.
Third vote passed 8/0/0, 12/3/98.

7.1 Introduction

This section describes routines which allow users to build point-to-point channels. Each point-to-point channel endpoint has an input buffer iterator and an output buffer iterator. The sending end of a channel takes buffers to be sent from its input iterator and puts them into its output iterator upon completion of message transfer. The receiving end of a channel takes buffers to be filled from its input iterator and puts them into its output iterator when the message transfer completes. Buffer iterators can be shared between channels.

> *Rationale.* Persistent point-to-point communications provide scheduled sends and receives, with specific expectations of both the synchronization and the ordering behavior between logical endpoints[1]. These persistent communication calls operate as pre-negotiated transfers, rather than as sends and receives that incidentally match by means of receipt selectivity (demultiplexing criteria). MPI/RT provides a way to bind such endpoints into point-to-point channels at the time channels are *committed*, as well as a more general strategy of defining a collective channel which describes a pattern of transfer involving more than a single point-to-point transfer. (*End of rationale.*)

[1] Compare this with MPI-1 persistent communications, defined locally as either sends or receives, but not across the process pair, which allow any receive to match a persistent send and any send to match a persistent receive [14]. This loose connection of the protocol endpoints obviates certain types of optimizations.

7.2 Operations on the Point-to-Point Channel Object

MPIRT_PTCHANNEL_CREATE (name, inbufiter, outbufiter, group, rank, direction, channel_qos, channel)

IN	name	name of the channel (MPIRT_STRING_NAME)
IN	inbufiter	input buffer iterator (MPIRT_BUFITER)
IN	outbufiter	output buffer iterator (MPIRT_BUFITER)
IN	group	group over which the channel operates (MPIRT_GROUP)
IN	rank	rank of the remote channel endpoint (non-negative integer)
IN	direction	endedness of the channel: MPIRT_CHANNEL_SEND for the sender, and MPIRT_CHANNEL_RECV for the receiver (MPIRT_CHANNEL_DIRECTION)
IN	channel_qos	quality-of-service (QoS) for the channel (MPIRT_QOS_CHANNEL) or MPIRT_QOS_IGNORE
OUT	channel	point-to-point channel (ref. to MPIRT_PTCHANNEL)

```
int MPIRT_Ptchannel_create (MPIRT_String_name name, MPIRT_Bufiter inbufiter,
              MPIRT_Bufiter outbufiter, MPIRT_Group group, int rank,
              MPIRT_Channel_direction direction, MPIRT_Qos_channel channel_qos,
              MPIRT_Ptchannel *channel)
```

```
MPIRT::Ptchannel (MPIRT::String_Name name, const MPIRT::Bufiter &inbufiter, const
              MPIRT::Bufiter &outbufiter, const MPIRT::Group &group, int rank,
              MPIRT::Channel::Direction direction, const
              MPIRT::Qos_Channel &channel_qos)
```

The descriptions of parameters name, channel_qos, group, inbufiter, and outbufiter, and operations for them, can be found in Chapter 6, since these parameters are common between point-to-point and collective channels. The operations of remaining parameters are defined below.

The following operations query rank and direction.

MPIRT_PTCHANNEL_GET_RANK (channel, rank)

IN	channel	point-to-point channel (MPIRT_PTCHANNEL)
OUT	rank	rank of the remote channel endpoint (ref. to non-negative integer)

```
int MPIRT_Ptchannel_get_rank (MPIRT_Ptchannel channel, int *rank)
```

```
int MPIRT::Ptchannel::Get_Rank (int &rank) const
```

MPIRT_PTCHANNEL_GET_DIRECTION (channel, direction)

IN	channel	point-to-point channel (MPIRT_PTCHANNEL)
OUT	direction	endedness of the channel: MPIRT_CHANNEL_SEND for the sender, and MPIRT_CHANNEL_RECV for the receiver (ref. to MPIRT_CHANNEL_DIRECTION)

```
int MPIRT_Ptchannel_get_direction (MPIRT_Ptchannel channel,
              MPIRT_Channel_direction *direction)
```

```
int MPIRT::Ptchannel::Get_Direction (MPIRT::Channel::Direction &direction) const
```

The following operations set rank and direction.

MPIRT_PTCHANNEL_SET_RANK (channel, rank)

INOUT	channel	point-to-point channel (ref. to MPIRT_PTCHANNEL)
IN	rank	rank of the remote channel endpoint (non-negative integer)

```
int MPIRT_Ptchannel_set_rank (MPIRT_Ptchannel *channel, int rank)
```

```
int MPIRT::Ptchannel::Set_Rank (int rank)
```

MPIRT_PTCHANNEL_SET_DIRECTION (channel, direction)

INOUT	channel	point-to-point channel (ref. to MPIRT_PTCHANNEL)
IN	direction	endedness of the channel: MPIRT_CHANNEL_SEND for the sender, and MPIRT_CHANNEL_RECV for the receiver (MPIRT_CHANNEL_DIRECTION)

```
int MPIRT_Ptchannel_set_direction (MPIRT_Ptchannel *channel,
              MPIRT_Channel_direction direction)
```

```
int MPIRT::Ptchannel::Set_Direction (MPIRT::Channel::Direction direction)
```

7.3 Summary

This section sumarizes enumerations and type declarations that are introduced in this chapter. ⊤ (final2)

⊥ (final2)

Miscellaneous Definitions		
Item	*What is it*	*Defined as:*
MPIRT_PTCHANNEL	point-to-point channel object	point-to-point channel object
MPIRT_OBJECT_TYPE_PTCHANNEL	a point-to-point channel's MPIRT_OBJECT_TYPE	enumeration
MPIRT_CHANNEL_SEND	send endpoint of a point-to-point channel	enumeration
MPIRT_CHANNEL_RECV	receive endpoint of a point-to-point channel	enumeration

Miscellaneous Bindings		
Item	*C Bindings*	*C++ Bindings*
MPIRT_PTCHANNEL	MPIRT_Ptchannel	MPIRT::Ptchannel
MPIRT_PTCHANNEL_NULL	MPIRT_PTCHANNEL_NULL	MPIRT::PTCHANNEL_NULL
MPIRT_OBJECT_TYPE_PTCHANNEL	MPIRT_OBJECT_TYPE_PTCHANNEL	MPIRT::OBJECT_TYPE_PTCHANNEL
MPIRT_CHANNEL_SEND	MPIRT_CHANNEL_SEND	MPIRT::Channel::SEND
MPIRT_CHANNEL_RECV	MPIRT_CHANNEL_RECV	MPIRT::Channel::RECV

Ptchannel Specific Operations		
Language Independent Specification	C	C++
MPIRT_PTCHANNEL_CREATE	MPIRT_Ptchannel_create	MPIRT::Ptchannel
MPIRT_PTCHANNEL_GET_RANK	MPIRT_Ptchannel_get_rank	MPIRT::Ptchannel::Get_Rank
MPIRT_PTCHANNEL_GET_DIRECTION	MPIRT_Ptchannel_get_direction	MPIRT::Ptchannel::Get_Direction
MPIRT_PTCHANNEL_SET_RANK	MPIRT_Ptchannel_set_rank	MPIRT::Ptchannel::Set_Rank
MPIRT_PTCHANNEL_SET_DIRECTION	MPIRT_Ptchannel_set_direction	MPIRT::Ptchannel::Set_Direction
MPIRT_PTCHANNEL_GET_INBUFITER	MPIRT_Ptchannel_get_inbufiter	MPIRT::Ptchannel::Get_Inbufiter
MPIRT_PTCHANNEL_GET_OUTBUFITER	MPIRT_Ptchannel_get_outbufiter	MPIRT::Ptchannel::Get_Outbufiter
MPIRT_PTCHANNEL_GET_GROUP	MPIRT_Ptchannel_get_group	MPIRT::Ptchannel::Get_Group
MPIRT_PTCHANNEL_GET_QOS	MPIRT_Ptchannel_get_qos	MPIRT::Ptchannel::Get_Qos
MPIRT_PTCHANNEL_SET_INBUFITER	MPIRT_Ptchannel_set_inbufiter	MPIRT::Ptchannel::Set_Inbufiter
MPIRT_PTCHANNEL_SET_OUTBUFITER	MPIRT_Ptchannel_set_outbufiter	MPIRT::Ptchannel::Set_Outbufiter
MPIRT_PTCHANNEL_SET_GROUP	MPIRT_Ptchannel_set_group	MPIRT::Ptchannel::Set_Group
MPIRT_PTCHANNEL_SET_QOS	MPIRT_Ptchannel_set_qos	MPIRT::Ptchannel::Set_Qos
MPIRT_PTCHANNEL_GET_NAME	MPIRT_Ptchannel_get_name	MPIRT::Ptchannel::Get_Name
MPIRT_PTCHANNEL_SET_NAME	MPIRT_Ptchannel_set_name	MPIRT::Ptchannel::Set_Name
MPIRT_PTCHANNEL_DUP	MPIRT_Ptchannel_dup	MPIRT::Ptchannel MPIRT::Ptchannel::Dup MPIRT::Ptchannel::Operator=
MPIRT_PTCHANNEL_FREE	MPIRT_Ptchannel_free	MPIRT::∼Ptchannel MPIRT::Ptchannel::Free
MPIRT_PTCHANNEL_IS_EQUAL	MPIRT_Ptchannel_is_equal	MPIRT::Ptchannel::Is_equal
MPIRT_PTCHANNEL_ATTACH_ATTR	MPIRT_Ptchannel_attach_attr	MPIRT::Ptchannel::Attach_Attr
MPIRT_PTCHANNEL_RETRIEVE_ATTR	MPIRT_Ptchannel_retrieve_attr	MPIRT::Ptchannel::Retrieve_Attr
MPIRT_PTCHANNEL_DELETE_ATTR	MPIRT_Ptchannel_delete_attr	MPIRT::Ptchannel::Delete_Attr

Chapter 8

Collective Channels

Current Status: First formal vote passed 7/0/0 on 5/20/98.
Second formal vote passed 9/0/2 on 7/98. Third formal vote passed 8/0/0 on 12/3/98.

8.1 Introduction

This chapter discusses the five basic collective operations supported by MPI/RT: barrier synchronization, scatter, gather, broadcast, and reduce. Each of these operations is performed over a collective channel created for these operations. This is analogous to the point-to-point channels for sends and receives. $\top$ (final2)

> *Advice to implementors.* The collective channel operations may be implemented using point-to-point channel operations. For each collective operation presented in this chapter, a sample implementation of the collective channel for this operation based on point-to-point channels and buffer iterators that obey the rules is presented. (*End of advice to implementors.*)

$\perp$ (final2)

The collective channels for these five operations share between themselves and with point-to-point channels the following attributes: name, channel_qos, group, inbufiter, and outbufiter. The descriptions of these parameters and operations on them can be found in Chapter 6. The other generic operations that are applicable to collective channels – dup, free, equality testing, $\top$ (final) and get/set name – can be found in Section 2.2. $\perp$ (final)

8.2 Broadcast Collective Channel

The operations for creating and manipulating the broadcast collective channel are defined in this section. Using a broadcast channel, a message from the process with rank *root* will be broadcast $\top$ (final) to all other processes of the group. $\perp$ (final)

MPIRT_BCAST_CHANNEL_CREATE (name, inbufiter, outbufiter, group, qos, root, broadcastchannel)

IN	name	name of the collective channel (MPIRT_STRING_NAME)
IN	inbufiter	input buffer iterator (MPIRT_BUFITER)
IN	outbufiter	output buffer iterator (MPIRT_BUFITER)
IN	group	group over which the collective channel operates (MPIRT_GROUP)
IN	qos	channel quality-of-service (QoS) (MPIRT_QOS_CHANNEL)
IN	root	rank of the broadcast source (non-negative integer)
OUT	broadcastchannel	broadcast channel (ref. to MPIRT_BCAST_CHANNEL)

```
int MPIRT_Bcast_channel_create (MPIRT_String_name name, MPIRT_Bufiter inbufiter,
            MPIRT_Bufiter outbufiter, MPIRT_Group group, MPIRT_Qos_channel qos,
            int root, MPIRT_Bcast_channel *broadcastchannel)
```

```
MPIRT::Bcast_Channel (MPIRT::String_Name name, const MPIRT::Bufiter &inbufiter,
            const MPIRT::Bufiter &outbufiter, const MPIRT::Group &group, const
            MPIRT::Qos_Channel &qos, int root)
```

Advice to implementors. This advice to implementors describes two examples a broadcast collective channel can be implemented using point-to-point channel constructs. Both examples create persistent point-to-point channels between the root and all other endpoints of the collective channel (members of the group). When the application calls the MPIRT_COMMIT operation, the example implementation creates these point-to-point channels with appropriate buffer iterators. These point-to-point channels are chained together, either explicitly by using handlers for the completion event of a message transfer over these point-to-point channels or implicitly such that the completion of the message transfer on one of the point-to-point channels automatically triggers the start of the message transfer over the next one until the message is transferred once over each of the point-to-point channels. The method for choosing these buffer iterators is described below.

The first example creates a single buffer iterator, of length one, of the same buffer type as the broadcast channel. This buffer iterator is both the input and output buffer iterator for all the point-to-point channels. When a message broadcast operation is issued over the broadcast channel, the first buffer is removed from the input buffer iterator of the collective channel and is inserted into this created buffer iterator. Then the chain of message transfers over all the created point-to-point channels starts. Upon completion of the message transfer over the last point-to-point channel, the buffer is inserted into the *out* buffer iterator of the broadcast channel.

The second example creates input buffer iterators and output buffer iterators for each point-to-point channel at the root process when the broadcast collective channel is committed. Only limited new memory (if at all) is allocated for the new buffer iterator since the buffers themselves were already created. The iterators are chained such that the first point-to-point channel's input buffer iterator is the input buffer of the broadcast collective channel. Its output buffer iterator is the second channel's shadow input buffer iterator. The output buffer iterator for the one point-to-point channel is the input buffer iterator for the next point-to-point channel. The last channel's output buffer iterator is the output buffer iterator for the broadcast collective channel. Figure 8.1 illustrates the shadow buffer iterator method.

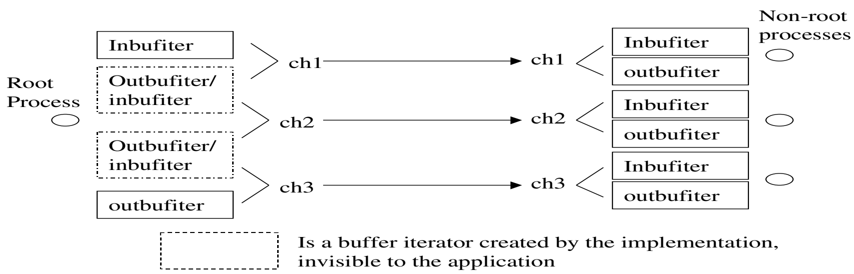

Figure 8.1: Point-to-point channel implementation example diagram for broadcast.

(*End of advice to implementors.*)

⊥ (final)

The following operations are specific to the broadcast channel object:

MPIRT_BCAST_CHANNEL_GET_ROOT (broadcastchannel, root)

IN	broadcastchannel	broadcast channel (MPIRT_BCAST_CHANNEL)
OUT	root	rank of the root endpoint (ref. to non-negative integer)

```
int MPIRT_Bcast_Channel_get_root (MPIRT_Bcast_Channel broadcastchannel, int *root)
```

```
MPIRT::Bcast_Channel::Get_Root (int &root) const
```

MPIRT_BCAST_CHANNEL_SET_ROOT (broadcastchannel, root)

INOUT	broadcastchannel	broadcast channel (ref. to MPIRT_BCAST_CHANNEL)
IN	root	rank of the root endpoint (non-negative integer)

```
int MPIRT_Bcast_Channel_set_root (MPIRT_Bcast_Channel *broadcastchannel, int root)
```

```
MPIRT::Bcast_Channel::Set_Root (int root)
```

8.3 Gather Operation Channel

This section defines operations for creating and manipulating parameters of a gather collective channel. In using the gather channel, each process (including the root process) sends the message through the out-buffer iterator to the root process. The arriving messages are concatenated together into the buffers of the receive-buffer iterator of the root process ordered by the rank of each participating process.

MPIRT_GATHER_CHANNEL_CREATE (name, inbufiter, outbufiter, recvinbufiter, recvoutbufiter, group, qos, root, gatherchannel)

IN	name	name of the collective channel (MPIRT_STRING_NAME)
IN	inbufiter	sender side input buffer iterator (MPIRT_BUFITER)
IN	outbufiter	sender side output buffer iterator (MPIRT_BUFITER)
IN	recvinbufiter	receiver side input buffer iterator that is significant only at the root process. For non-root processes, its value can be MPIRT_BUFITER_NULL (MPIRT_BUFITER)
IN	recvoutbufiter	receiver side output buffer iterator that is significant only at the root process. For non-root processes, its value can be MPIRT_BUFITER_NULL (MPIRT_BUFITER)
IN	group	group over which the collective channel operates (MPIRT_GROUP)
IN	qos	channel QoS (MPIRT_QOS_CHANNEL)
IN	root	rank of the gather root process (non-negative integer)
OUT	gatherchannel	gather collective channel (ref. to MPIRT_GATHER_CHANNEL)

```
int MPIRT_Gather_channel_create (MPIRT_String_name, MPIRT_Bufiter inbufiter,
            MPIRT_Bufiter outbufiter, MPIRT_Bufiter recvinbufiter,
            MPIRT_Bufiter recvoutbufiter, MPIRT_Group group,
            MPIRT_Qos_channel qos, int root, MPIRT_Gather_channel *gatherchannel
```

```
int MPIRT::Gather_Channel (MPIRT::String_Name name, const
            MPIRT::Bufiter &inbufiter, const MPIRT::Bufiter &outbufiter, const
            MPIRT::Bufiter &recvinbufiter, const MPIRT::Bufiter &recvoutbufiter,
            const MPIRT::Group &group, const MPIRT::Qos_Channel &qos, int root)
```

By convention, inbufiter and outbufiter represent send side buffer iterators (see Chapter 6). A buffer for the receiver side buffer iterator that is defined only for the root process is of the same dataspec as sender side buffers and is of size equal to the number of senders of the gather channel (the size of the group) times the size of the send buffer.

Advice to implementors. This advice to implementors describes an example how to implement a gather collective channel using point-to-point channels. Persistent point-to-point channels are created between the root and all sending endpoints of the gather collective channel (members of the group including the root itself). When the application calls the

MPIRT_COMMIT operation, the example implementation creates these point-to-point channels with appropriate buffer iterators. The send sides of the point-to-point channels share the buffer iterators with the collective gather channel send side. Once a send side of a gather collective channel is started, the example implementation starts the send side of a point-to-point channel. This removes the buffer from inbufiter that is shared between the collective gather channel and the point-to-point channel. Upon completion, the example implementation inserts this buffer into the outbufiter that is shared between the collective gather channel and the point-to-point channel.

For the root, the following strategy defines the receiving side buffer iterators for all these point-to-point channels. For each point-to-point channel, the input receive buffer iterator will be of the same size (maxbufcount) as recvinbufiter, and the output buffer iterator will be of size (maxbufcount) of recvoutbufiter. This will allow the point-to-point channels to support the same number of buffers as the collective gather channel. The dataspecs of all the buffers in all buffer iterators for both the send and receive sides and the *in* and *out* buffer iterators are all the same. The size of the buffers in the *in* and *out* buffer iterators for point-to-point channels is the same as for the sending side. The combined length of single buffers for all receive side buffer iterators for all these point-to-point channels is exactly equal to the size of a single buffer of the receive side buffer iterator of the collective gather channel. A buffer in the recvinbufiter is split into n equal parts, where the number of parts equals the number of endpoints of the gather collective channel. That means that a "big" buffer in the gather collective channel buffer iterator is a concatenation of the individual point-to-point channel buffers that share the memory with the "big" buffer. For each "big" buffer in the recvinbufiter, there is a corresponding single "small" buffer in the input buffer iterator for the receive side of each point-to-point channel.

When the message transfer starts over the collective gather channel, it starts simultaneous message transfers on all point-to-point channels. Every operation on the "big" buffer of the receive side buffer iterator for the collective gather channel is synchronized with an operation for each of the point-to-point channel's receive side buffer iterators. When the buffer is removed from the recvinbufiter for the collective channel, the corresponding buffer is removed from the *in* buffer iterator for the receive side of all point-to-point channels (at the root). When all the point-to-point channels have received their messages and have inserted their buffers into their appropriate *out* buffer iterators, then their concatenation that is the "big" buffer is inserted into the recvoutbufiter for the collective gather channel. The example implementation can either use a container for all point-to-point channels to start all of them and wait for their completion or use handlers for their completion events in order to keep track of their statuses. When users remove a buffer from recvoutbufiter, the example implementation automatically removes a buffer from the *out* buffer iterator for each of the point-to-point channels. Analogously, when users insert a buffer into recvinbufiter, the example implementation automatically inserts a buffer into the *in* buffer iterator for a receive side of each of the point-to-point channels. Figure 8.2 shows the point-to-point channel strategy for collective gather channels.

(*End of advice to implementors.*)

⊥ (final)

The following operations query and set attributes of the gather collective channel.

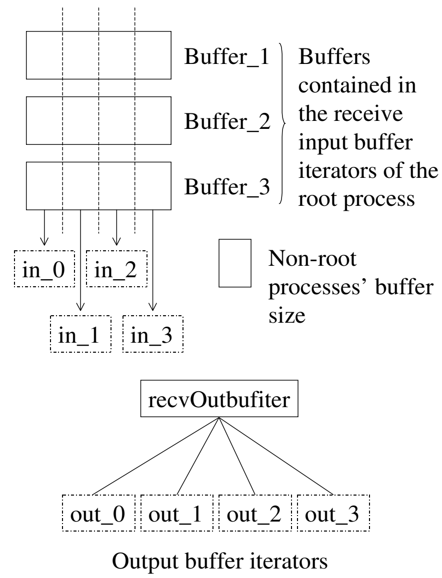

Figure 8.2: Point-to-point channels' scheme for implementing gather operation.

MPIRT_GATHER_CHANNEL_GET_RECVINBUFITER (gatherchannel, recvinbufiter)

IN	gatherchannel	gather collective channel (MPIRT_GATHER_CHANNEL)
OUT	recvinbufiter	either MPIRT_BUFITER_NULL for a non-root process, or the receive side input buffer iterator for the root (ref. to MPIRT_BUFITER)

```
int MPIRT_Gather_channel_get_recvinbufiter (MPIRT_Gather_channel gatherchannel,
            MPIRT_Bufiter *recvinbufiter)
```

```
int MPIRT::Gather_Channel::Get_Recvinbufiter (MPIRT::Bufiter *&recvinbufiter)
            const
```

MPIRT_GATHER_CHANNEL_GET_RECVOUTBUFITER (gatherchannel, recvoutbufiter)

IN	gatherchannel	gather collective channel (MPIRT_GATHER_CHANNEL)
OUT	recvoutbufiter	either MPIRT_BUFITER_NULL for a non-root process, or the receive side output buffer iterator for the root (ref. to MPIRT_BUFITER)

```
int MPIRT_Gather_channel_get_recvoutbufiter (MPIRT_Gather_channel gatherchannel,
            MPIRT_Bufiter *recvoutbufiter)
```

```
int MPIRT::Gather_Channel::Get_Recvoutbufiter (MPIRT::Bufiter *&recvoutbufiter)
              const
```

The following function returns the rank of the root process.

MPIRT_GATHER_CHANNEL_GET_ROOT (gatherchannel, root)

IN	gatherchannel	gather collective channel (MPIRT_GATHER_CHANNEL)
OUT	root	rank of the root endpoint (non-negative integer)

```
int MPIRT_Gather_channel_get_root (MPIRT_Gather_channel gatherchannel, int *root)
```

```
int MPIRT::Gather_Channel::Get_Root (int &root) const
```

MPIRT_GATHER_CHANNEL_SET_RECVINBUFITER (gatherchannel, recvinbufiter)

INOUT	gatherchannel	gather collective channel (ref. to MPIRT_GATHER_CHANNEL)
IN	recvinbufiter	either MPIRT_BUFITER_NULL for a non-root process, or the receive side input buffer iterator for the root (MPIRT_BUFITER)

```
int MPIRT_Gather_channel_set_recvinbufiter (MPIRT_Gather_channel *gatherchannel,
              MPIRT_Bufiter recvinbufiter)
```

```
int MPIRT::Gather_Channel::Set_Recvinbufiter (const MPIRT::Bufiter &recvinbufiter)
```

MPIRT_GATHER_CHANNEL_SET_RECVOUTBUFITER (gatherchannel, recvoutbufiter)

INOUT	gatherchannel	gather collective channel (ref. to MPIRT_GATHER_CHANNEL)
IN	recvoutbufiter	either MPIRT_BUFITER_NULL for a non-root process, or the receive side output buffer iterator for the root (MPIRT_BUFITER)

```
int MPIRT_Gather_channel_set_recvoutbufiter (MPIRT_Gather_channel *gatherchannel,
              MPIRT_Bufiter recvoutbufiter)
```

```
int MPIRT::Gather_Channel::Set_Recvoutbufiter (const
              MPIRT::Bufiter &recvoutbufiter)
```

The following function sets the rank of the root process for the gather channel.

MPIRT_GATHER_CHANNEL_SET_ROOT (gatherchannel, root)

INOUT	gatherchannel	gather collective channel (ref. to MPIRT_GATHER_CHANNEL)
IN	root	rank of the root endpoint (non-negative integer)

```
int MPIRT_Gather_channel_set_root (MPIRT_Gather_channel *gatherchannel, int root)
```

```
int MPIRT::Gather_Channel::Set_Root (int root)
```

8.4 Scatter Operation Channel

This section defines operations for creating and manipulating a scatter collective channel. The scatter operation is the inverse operation of a gather operation. The message buffer in the out buffer iterator of the root process will be evenly divided into small buffers with the number equal to the size of the group. Each small buffer will be sent out to the corresponding receive process in terms of the process's rank.

MPIRT_SCATTER_CHANNEL_CREATE (name, inbufiter, outbufiter, recvinbufiter, recvoutbufiter, group, qos, root, scatterchannel)

IN	name	name of the collective channel (MPIRT_STRING_NAME)
IN	inbufiter	sender side input buffer iterator that is significant only at the root process, MPIRT_BUFITER_NULL for non-root processes (MPIRT_BUFITER)
IN	outbufiter	sender side output buffer iterator that is significant only at the root process, MPIRT_BUFITER_NULL for non-root processes (MPIRT_BUFITER)
IN	recvinbufiter	receiver side input buffer iterator (MPIRT_BUFITER)
IN	recvoutbufiter	receiver side output buffer iterator (MPIRT_BUFITER)
IN	group	group over which the collective channel operates (MPIRT_GROUP)
IN	qos	channel QoS (MPIRT_QOS_CHANNEL)
IN	root	rank of the scatter source (non-negative integer)
OUT	scatterchannel	scatter collective channel (ref. to MPIRT_SCATTER_CHANNEL)

```
int MPIRT_Scatter_channel_create (MPIRT_String_name name, MPIRT_Bufiter inbufiter,
              MPIRT_Bufiter outbufiter, MPIRT_Bufiter recvinbufiter,
              MPIRT_Bufiter recvoutbufiter, MPIRT_Group group,
              MPIRT_Qos_channel qos, int root,
              MPIRT_Scatter_channel *scatterchannel)
```

```
MPIRT::Scatter_Channel (MPIRT::String_Name name, const MPIRT::Bufiter &inbufiter,
              const MPIRT::Bufiter &outbufiter, const
```

```
              MPIRT::Bufiter &recvinbufiter, const MPIRT::Bufiter &recvoutbufiter,
              const MPIRT::Group &group, const MPIRT::Qos_Channel &qos, int root)
```

By convention, inbufiter and outbufiter represent send side buffer iterators (see Chapter 6). The send side buffer iterators are defined only for root process and are MPIRT_BUFITER_NULL for the non-root processes of the scatter channel. A buffer for the send side buffer iterator is of the same dataspec as the receiver side buffers and is of size equal to the number of receivers of the scatter channel (the size of the group) times the size of the receiver buffer.

> *Advice to implementors.* An example of the implementation for the scatter collective channel is similar to the gather collective channel using point-to-point channels except that the role of send-side and receive-side buffer iterators is reversed. (*End of advice to implementors.*)

The next six functions query and set individual attributes of the scatter collective channel.

MPIRT_SCATTER_CHANNEL_GET_RECVINBUFITER (scatterchannel, recvinbufiter)

IN	scatterchannel	scatter collective channel (MPIRT_SCATTER_CHANNEL)
OUT	recvinbufiter	receiver side input buffer iterator that is significant only at the root process. For non-root processes, its value can be MPIRT_BUFITER_NULL (ref. to MPIRT_BUFITER)

```
int MPIRT_Scatter_channel_get_recvinbufiter (MPIRT_Scatter_channel scatterchannel,
              MPIRT_Bufiter *recvinbufiter)
```

```
int MPIRT::Scatter_Channel::Get_Recvinbufiter (MPIRT::Bufiter *&recvinbufiter)
              const
```

MPIRT_SCATTER_CHANNEL_GET_RECVOUTBUFITER (scatterchannel, recvoutbufiter)

IN	scatterchannel	scatter collective channel (MPIRT_SCATTER_CHANNEL)
OUT	recvoutbufiter	receiver side output buffer iterator that is significant only at the root process. For non-root processes, its value can be MPIRT_BUFITER_NULL (ref. to MPIRT_BUFITER)

```
int MPIRT_Scatter_channel_get_recvoutbufiter (MPIRT_Scatter_channel Scatterchannel,
              MPIRT_Bufiter *recvoutbufiter)
```

```
int MPIRT::Scatter_Channel::Get_Recvoutbufiter (MPIRT::Bufiter *&recvoutbufiter)
              const
```

MPIRT_SCATTER_CHANNEL_GET_ROOT (scatterchannel, root)

IN	scatterchannel	scatter collective channel (MPIRT_SCATTER_CHANNEL)
OUT	root	rank of the root endpoint (ref. to non-negative integer)

```
int MPIRT_Scatter_channel_get_root (MPIRT_Scatter_channel scatterchannel,
            int *root)
```

```
int MPIRT::Scatter_Channel::Get_Root (int &root) const
```

MPIRT_SCATTER_CHANNEL_SET_RECVINBUFITER (scatterchannel, recvinbufiter)

INOUT	scatterchannel	scatter collective channel (ref. to MPIRT_SCATTER_CHANNEL)
IN	recvinbufiter	receiver side input buffer iterator that is significant only at the root process. For non-root processes, its value can be MPIRT_BUFITER_NULL (MPIRT_BUFITER)

```
int MPIRT_Scatter_channel_set_recvinbufiter (MPIRT_Scatter_channel *scatterchannel,
            MPIRT_Bufiter recvinbufiter)
```

```
int MPIRT::Scatter_Channel::Set_Recvinbufiter (const
            MPIRT::Bufiter &recvinbufiter)
```

MPIRT_SCATTER_CHANNEL_SET_RECVOUTBUFITER (scatterchannel, recvoutbufiter)

INOUT	scatterchannel	scatter collective channel (ref. to MPIRT_SCATTER_CHANNEL)
IN	recvoutbufiter	receiver side output buffer iterator that is significant only at the root process. For non-root processes, its value can be MPIRT_BUFITER_NULL (MPIRT_BUFITER)

```
int MPIRT_Scatter_channel_set_recvoutbufiter
            (MPIRT_Scatter_channel *scatterchannel, MPIRT_Bufiter recvoutbufiter)
```

```
int MPIRT::Scatter_Channel::Set_Recvoutbufiter (const
            MPIRT::Bufiter &recvoutbufiter)
```

MPIRT_SCATTER_CHANNEL_SET_ROOT (scatterchannel, root)

INOUT	scatterchannel	scatter collective channel (ref. to MPIRT_SCATTER_CHANNEL)
IN	root	rank of the root endpoint (non-negative integer)

```
int MPIRT_Scatter_channel_set_root (MPIRT_Scatter_channel *scatterchannel,
            int root)
```

```
int MPIRT::Scatter_Channel::Set_Root (int root)
```

8.5 Reduce Operation Channel

The function defined below creates a reduce collective channel. The reduce operation is very similar to the gather operation, except that instead of concatenating the buffers sent by the participating processes, a reduce operation is performed on all the incoming data, and the result is stored into a receive buffer at the root side.

MPIRT_REDUCE_CHANNEL_CREATE (name, inbufiter, outbufiter, recvinbufiter, recvoutbufiter, op, group, qos, root, reducechannel)

IN	name	name of the collective channel (MPIRT_STRING_NAME)
IN	inbufiter	sender side input buffer iterator (MPIRT_BUFITER)
IN	outbufiter	sender side output buffer iterator (MPIRT_BUFITER)
IN	recvinbufiter	receiver side input buffer iterator that is significant only at the root process and MPIRT_BUFITER_NULL for non-root processes (MPIRT_BUFITER)
IN	recvoutbufiter	receiver side output buffer iterator object that is significant only at the root process and MPIRT_BUFITER_NULL for non-root processes (MPIRT_BUFITER) ⊤ (final) ⊥ (final)
IN	op	reduce operation (MPIRT_OP)
IN	group	group over which the reduce collective channel operates (MPIRT_GROUP)
IN	qos	channel QoS (MPIRT_QOS_CHANNEL)
IN	root	rank of the reduce root process (non-negative integer)
OUT	reducechannel	reduce collective channel (ref. to MPIRT_REDUCE_CHANNEL)

```
int MPIRT_Reduce_channel_create (MPIRT_String_name name, MPIRT_Bufiter inbufiter,
            MPIRT_Bufiter outbufiter, MPIRT_Bufiter recvinbufiter,
            MPIRT_Bufiter recvoutbufiter, MPIRT_Op op, MPIRT_Group group,
            MPIRT_Qos_channel qos, int root, MPIRT_Reduce_channel *reducechannel)
```

```
MPIRT::Reduce_Channel (MPIRT::String_Name name, const MPIRT::Bufiter &inbufiter,
            const MPIRT::Bufiter &outbufiter, const
            MPIRT::Bufiter &recvinbufiter, const MPIRT::Bufiter &recvoutbufiter,
            MPIRT::Op op, const MPIRT::Group &group, const
            MPIRT::Qos_Channel &qos, int root)
```

⊤ (final)

Advice to implementors. The example implementation of the reduce collective channel using point-to-point channels is similar to the point-to-point implementation of the gather channel. The difference is that receive buffer iterators for the point-to-point channels need to be created so that they do not necessarily use the reduce collective channel receive side buffers (this is reduce operation dependent). The example implementation can create a single buffer iterator for the receive side of the point-to-point channels and allocate memory for them at commit time. Upon receiving the messages from all the receiving point-to-point channels, it can perform the reduce operation after removing all the buffers from the *out* buffer iterators and write the result into the reduce collective channel buffer. Alternatively, it can set up handlers to partially perform the reduce operation, which will be invoked upon message reception on point-to-point channels. The handlers can be shared among all the point-to-point channels (a single handler for event receptors for all channels' completion events). The handler can even perform the reduce operation in the reduce collective channel buffer. When all point-to-point channels finish message transfers, the example implementation will insert the buffer into recvoutbufiter.

Alternatively, the example implementation can follow the broadcast implementation approach and order the point-to-point channels appropriately. This way no new buffers need to be allocated by the example implementation. It can use the same receive side buffer as the reduce collective channel in which all the point-to-point channels will receive the message. The example implementation will still need to store partial results so that the next point-to-point channel message arrival does not destroy the previous partial results.

All buffers in the send buffer iterator and receive buffer iterators for the point-to-point channels have the same size. At commit time, the example implementation creates point-to-point channels and shadow buffers just as for the gather operation. (*End of advice to implementors.*)

⊥ (final)

The next eight operations query and set attributes of the reduce collective channel.

MPIRT_REDUCE_CHANNEL_GET_RECVINBUFITER (reducechannel, recvinbufiter)

IN	reducechannel	reduce collective channel (MPIRT_REDUCE_CHANNEL)
OUT	recvinbufiter	receiver side input buffer iterator that is significant only at the root process and MPIRT_BUFITER_NULL for non-root processes (ref. to MPIRT_BUFITER)

```
int MPIRT_Reduce_channel_get_recvinbufiter (MPIRT_Reduce_channel reducechannel,
            MPIRT_Bufiter *recvinbufiter)
```

```
int MPIRT::Reduce_Channel::Get_Recvinbufiter (MPIRT::Bufiter *&recvinbufiter)
            const
```

MPIRT_REDUCE_CHANNEL_GET_RECVOUTBUFITER (reducechannel, recvoutbufiter)

IN	reducechannel	reduce collective channel (MPIRT_REDUCE_CHANNEL)
OUT	recvoutbufiter	receiver side output buffer iterator that is significant only at the root process and MPIRT_BUFITER_NULL for non-root processes (ref. to MPIRT_BUFITER)

```
int MPIRT_Reduce_channel_get_recvoutbufiter (MPIRT_Reduce_channel Reducechannel,
            MPIRT_Bufiter *recvoutbufiter)
```

```
int MPIRT::Reduce_Channel::Get_Recvoutbufiter (MPIRT::Bufiter *&recvoutbufiter)
            const
```

The following function returns the reduce operator.

MPIRT_REDUCE_CHANNEL_GET_OP (reducechannel, op)

IN	reducechannel	reduce collective channel (MPIRT_REDUCE_CHANNEL)
OUT	op	reduce operation (ref. to MPIRT_OP)

```
int MPIRT_Reduce_channel_get_op (MPIRT_Reduce_channel reducechannel, MPIRT_op *op)
```

```
int MPIRT::Reduce_Channel::Get_Op (MPIRT::Op &op) const
```

MPIRT_REDUCE_CHANNEL_GET_ROOT (reducechannel, root)

IN	reducechannel	reduce collective channel (MPIRT_REDUCE_CHANNEL)
OUT	root	rank of the root endpoint (ref. to non-negative integer)

```
int MPIRT_Reduce_channel_get_root (MPIRT_Reduce_channel reducechannel, int *root)
```

```
int MPIRT::Reduce_Channel::Get_Root (int &root) const
```

MPIRT_REDUCE_CHANNEL_SET_RECVINBUFITER (reducechannel, recvinbufiter)

INOUT	reducechannel	reduce collective channel (ref. to MPIRT_REDUCE_CHANNEL)
IN	recvinbufiter	receiver side input buffer iterator that is significant only at the root process and MPIRT_BUFITER_NULL for non-root processes (MPIRT_BUFITER)

```
int MPIRT_Reduce_channel_set_recvinbufiter (MPIRT_Reduce_channel *reducechannel,
            MPIRT_Bufiter recvinbufiter)
```

```
int MPIRT::Reduce_Channel::Set_Recvinbufiter (const MPIRT::Bufiter &recvinbufiter)
```

MPIRT_REDUCE_CHANNEL_SET_RECVOUTBUFITER (reducechannel, recvoutbufiter)

INOUT	reducechannel	reduce collective channel (ref. to MPIRT_REDUCE_CHANNEL)
IN	recvoutbufiter	receiver side output buffer iterator that is significant only at the root process and MPIRT_BUFITER_NULL for non-root processes (MPIRT_BUFITER)

```
int MPIRT_Reduce_channel_set_recvoutbufiter (MPIRT_Reduce_channel *reducechannel,
            MPIRT_Bufiter recvoutbufiter)
```

```
int MPIRT::Reduce_Channel::Set_Recvoutbufiter (const
            MPIRT::Bufiter &recvoutbufiter)
```

The following function sets the reduce operator.

MPIRT_REDUCE_CHANNEL_SET_OP (reducechannel, op)

INOUT	reducechannel	reduce collective channel (ref. to MPIRT_REDUCE_CHANNEL)
IN	op	reduce operation (MPIRT_OP)

```
int MPIRT_Reduce_channel_set_op (MPIRT_Reduce_channel *reducechannel, MPIRT_Op op)
```

```
int MPIRT::Reduce_Channel::Set_Op (MPIRT::Op op)
```

MPIRT_REDUCE_CHANNEL_SET_ROOT (reducechannel, root)

INOUT	reducechannel	reduce collective channel (ref. to MPIRT_REDUCE_CHANNEL)
IN	root	rank of the root endpoint (non-negative integer)

```
int MPIRT_Reduce_channel_set_root (MPIRT_Reduce_channel *reducechannel, int root)
```

```
int MPIRT::Reduce_Channel::Set_Root (int root)
```

8.5.1 Predefined Reduce Operations

The following predefined operations are supplied for the MPI/RT reduce operations (see Table 8.1). They are used by setting the op attribute of the reduce collective channel to one of the following values.

Operations	*Definitions*	*types operated on*
MPIRT_MAX	maximum	Integer, Floating point
MPIRT_MIN	minimum	Integer, Floating point
MPIRT_SUM	sum	Integer, Floating point
MPIRT_PROD	product	Integer, Floating point
MPIRT_LAND	logical *and*	Integer, Boolean
MPIRT_BAND	bit-wise *and*	Integer, Byte
MPIRT_LOR	logical *or*	Integer, Boolean
MPIRT_BOR	bit-wise *or*	Integer, Byte
MPIRT_LXOR	logical *xor*	Integer, Boolean
MPIRT_BXOR	bit-wise *xor*	Integer, Byte

Table 8.1: MPI/RT predefined reduce operations.

Table 8.2 shows the MPI/RT dataspecs over which reduce operations are defined. Compatible types are grouped as rows in the table.

Integer	MPIRT_DATASPEC_INT, MPIRT_DATASPEC_LONG, MPIRT_DATASPEC_SHORT, MPIRT_DATASPEC_UNSIGNED_SHORT, MPIRT_DATASPEC_UNSIGNED, MPIRT_DATASPEC_UNSIGNED_LONG
Floating point	MPIRT_DATASPEC_FLOAT, MPIRT_DATASPEC_DOUBLE, MPIRT_DATASPEC_LONG_DOUBLE
Logical	MPIRT_DATASPEC_BOOLEAN
Byte	MPIRT_DATASPEC_BYTE

Table 8.2: Types on which reduce operations are defined.

8.6 Barrier Operation Channel

This section describes the barrier synchronization operation collective channel. A process executing a barrier synchronization operation is blocked until each of the other participating processes has also executed the barrier operation.

The following function creates a barrier collective channel object. The input buffer iterator and output buffer iterator can be set to MPIRT_BUFITER_NULL.

MPIRT_BARRIER_CHANNEL_CREATE (name, inbufiter, outbufiter, group, qos, barrierchannel)

IN	name	name of the collective channel (MPIRT_STRING_NAME)
IN	inbufiter	input buffer iterator (MPIRT_BUFITER)
IN	outbufiter	output buffer iterator (MPIRT_BUFITER)
IN	group	group over which the collective channel operates (MPIRT_GROUP)
IN	qos	channel QoS (MPIRT_QOS_CHANNEL)
OUT	barrierchannel	barrier channel object (ref. to MPIRT_BARRIER_CHANNEL)

```
int MPIRT_Barrier_channel_create (MPIRT_String_name name, MPIRT_BUFITER inbufiter,
             MPIRT_BUFITER outbufiter, MPIRT_Group group,
             MPIRT_Qos_channel qos,MPIRT_Channel_barrier *barrierchannel)
```

```
MPIRT::Barrier_Channel (MPIRT::String_Name name, const MPIRT::Bufiter &inbufiter,
             const MPIRT::Bufiter &outbufiter, const MPIRT::Group &group, const
             MPIRT::Qos_Channel &qos)
```

⊤ (final)

Rationale. The buffer iterators are defined for this operation to maintain synchronization between an application and an implementation. Also the buffer iterators follow a standard format of all other channels, hence it inherits all the channel operations and the barrier channel follows the stata transition diagram of all other channel. (*End of rationale.*)

Advice to users. The buffer iterators are here only for synchronization purposes and to maintain the state transition diagram. Users can define the same buffer iterator for both in and out-buffer iterators with a single buffer of zero size, and it will be sufficient for this synchronization purpose. (*End of advice to users.*)

Advice to implementors. A sample implementation can elect a root and create one gather and one broadcast channel with zero size buffers. Once the gather operation completes, the root broadcasts to all group members that a synchronization point is reached. (*End of advice to implementors.*)

⊥ (final)

There are no operations specific to barrier collective channels.

8.7 Summary

This section summarizes enumerations and type declarations that are introduced in this chapter.

Miscellaneous Definitions		
Item	*What is it*	*Defined as:*
MPIRT_BCAST_CHANNEL	broadcast collective channel	broadcast collective channel
MPIRT_GATHER_CHANNEL	gather collective channel	gather collective channel
MPIRT_SCATTER_CHANNEL	scatter collective channel	scatter collective channel
MPIRT_REDUCE_CHANNEL	reduce collective channel	reduce collective channel
MPIRT_BARRIER_CHANNEL	barrier collective channel	barrier collective channel
MPIRT_OBJECT_TYPE_BCAST_CHANNEL	a broadcast channel's MPIRT_OBJECT_TYPE	enumeration
MPIRT_OBJECT_TYPE_GATHER_CHANNEL	a gather channel's MPIRT_OBJECT_TYPE	enumeration
MPIRT_OBJECT_TYPE_SCATTER_CHANNEL	a scatter channel's MPIRT_OBJECT_TYPE	enumeration
MPIRT_OBJECT_TYPE_REDUCE_CHANNEL	a reduce channel's MPIRT_OBJECT_TYPE	enumeration
MPIRT_OBJECT_TYPE_BARRIER_CHANNEL	a barrier channel's MPIRT_OBJECT_TYPE	enumeration

Collective Channel Reduce Operations		
Operations	*Definitions*	*types operated on*
MPIRT_MAX	maximum	Integer, Floating point
MPIRT_MIN	minimum	Integer, Floating point
MPIRT_SUM	sum	Integer, Floating point
MPIRT_PROD	product	Integer, Floating point
MPIRT_LAND	logical *and*	Integer, Boolean
MPIRT_BAND	bit-wise *and*	Integer, Byte
MPIRT_LOR	logical *or*	Integer, Boolean
MPIRT_BOR	bit-wise *or*	Integer, Byte
MPIRT_LXOR	logical *xor*	Integer, Boolean
MPIRT_BXOR	bit-wise *xor*	Integer, Byte

⊤ (final2)

⊥ (final2)

Miscellaneous Bindings		
Item	*C Bindings*	*C++ Bindings*
MPIRT_BCAST_CHANNEL	MPIRT_Bcast_channel	MPIRT::Bcast_Channel
MPIRT_GATHER_CHANNEL	MPIRT_Gather_channel	MPIRT::Gather_Channel
MPIRT_SCATTER_CHANNEL	MPIRT_Scatter_channel	MPIRT::Scatter_Channel
MPIRT_REDUCE_CHANNEL	MPIRT_Reduce_channel	MPIRT::Reduce_Channel
MPIRT_BARRIER_CHANNEL	MPIRT_Barrier_channel	MPIRT::Barrier_Channel
MPIRT_BCAST_CHANNEL_NULL	MPIRT_BCAST_CHANNEL_NULL	MPIRT::BCAST_CHANNEL_NULL
MPIRT_GATHER_CHANNEL_NULL	MPIRT_GATHER_CHANNEL_NULL	MPIRT::GATHER_CHANNEL_NULL
MPIRT_SCATTER_CHANNEL_NULL	MPIRT_SCATTER_CHANNEL_NULL	MPIRT::SCATTER_CHANNEL_NULL
MPIRT_REDUCE_CHANNEL_NULL	MPIRT_REDUCE_CHANNEL_NULL	MPIRT::REDUCE_CHANNEL_NULL
MPIRT_BARRIER_CHANNEL_NULL	MPIRT_BARRIER_CHANNEL_NULL	MPIRT::BARRIER_CHANNEL_NULL
MPIRT_OBJECT_TYPE_BCAST_CHANNEL	MPIRT_OBJECT_TYPE_BCAST_CHANNEL	MPIRT::OBJECT_TYPE_BCAST_CHANNEL
MPIRT_OBJECT_TYPE_GATHER_CHANNEL	MPIRT_OBJECT_TYPE_GATHER_CHANNEL	MPIRT::OBJECT_TYPE_GATHER_CHANNEL
MPIRT_OBJECT_TYPE_SCATTER_CHANNEL	MPIRT_OBJECT_TYPE_SCATTER_CHANNEL	MPIRT::OBJECT_TYPE_SCATTER_CHANNEL
MPIRT_OBJECT_TYPE_REDUCE_CHANNEL	MPIRT_OBJECT_TYPE_REDUCE_CHANNEL	MPIRT::OBJECT_TYPE_REDUCE_CHANNEL
MPIRT_OBJECT_TYPE_BARRIER_CHANNEL	MPIRT_OBJECT_TYPE_BARRIER_CHANNEL	MPIRT::OBJECT_TYPE_BARRIER_CHANNEL
MPIRT_MAX	MPIRT_MAX	MPIRT::Reduce_Channel::MAX
MPIRT_MIN	MPIRT_MIN	MPIRT::Reduce_Channel::MIN
MPIRT_SUM	MPIRT_SUM	MPIRT::Reduce_Channel::SUM
MPIRT_PROD	MPIRT_PROD	MPIRT::Reduce_Channel::PROD
MPIRT_LAND	MPIRT_LAND	MPIRT::Reduce_Channel::LAND
MPIRT_BAND	MPIRT_BAND	MPIRT::Reduce_Channel::BAND
MPIRT_LOR	MPIRT_LOR	MPIRT::Reduce_Channel::LOR
MPIRT_BOR	MPIRT_BOR	MPIRT::Reduce_Channel::BOR
MPIRT_LXOR	MPIRT_LXOR	MPIRT::Reduce_Channel::LXOR
MPIRT_BXOR	MPIRT_BXOR	MPIRT::Reduce_Channel::BXOR

Broadcast Channel Specific Operations		
Language Independent Specification	C	C++
MPIRT_BCAST_CHANNEL_CREATE	MPIRT_Bcast_channel_create	MPIRT::Bcast_Channel
MPIRT_BCAST_CHANNEL_GET_ROOT	MPIRT_Bcast_channel_get_root	MPIRT::Bcast_Channel::Get_Root
MPIRT_BCAST_CHANNEL_SET_ROOT	MPIRT_Bcast_channel_set_root	MPIRT::Bcast_Channel::Set_Root
MPIRT_BCAST_CHANNEL_GET_INBUFITER	MPIRT_Bcast_channel_get_inbufiter	MPIRT::Bcast_Channel::Get_Inbufiter
MPIRT_BCAST_CHANNEL_GET_OUTBUFITER	MPIRT_Bcast_channel_get_outbufiter	MPIRT::Bcast_Channel::Get_Outbufiter
MPIRT_BCAST_CHANNEL_GET_GROUP	MPIRT_Bcast_channel_get_group	MPIRT::Bcast_Channel::Get_Group
MPIRT_BCAST_CHANNEL_GET_QOS	MPIRT_Bcast_channel_get_qos	MPIRT::Bcast_Channel::Get_Qos
MPIRT_BCAST_CHANNEL_SET_INBUFITER	MPIRT_Bcast_channel_set_inbufiter	MPIRT::Bcast_Channel::Set_Inbufiter
MPIRT_BCAST_CHANNEL_SET_OUTBUFITER	MPIRT_Bcast_channel_set_outbufiter	MPIRT::Bcast_Channel::Set_Outbufiter
MPIRT_BCAST_CHANNEL_SET_GROUP	MPIRT_Bcast_channel_set_group	MPIRT::Bcast_Channel::Set_Group
MPIRT_BCAST_CHANNEL_SET_QOS	MPIRT_Bcast_channel_set_qos	MPIRT::Bcast_Channel::Set_Qos
MPIRT_BCAST_CHANNEL_GET_NAME	MPIRT_Bcast_channel_get_name	MPIRT::Bcast_Channel::Get_Name
MPIRT_BCAST_CHANNEL_SET_NAME	MPIRT_Bcast_channel_set_name	MPIRT::Bcast_Channel::Set_Name
MPIRT_BCAST_CHANNEL_DUP	MPIRT_Bcast_channel_dup	MPIRT::Bcast_Channel MPIRT::Bcast_Channel::Dup MPIRT::Bcast_Channel::Operator=
MPIRT_BCAST_CHANNEL_FREE	MPIRT_Bcast_channel_free	MPIRT::∼Bcast_Channel MPIRT::Bcast_Channel::Free
MPIRT_BCAST_CHANNEL_IS_EQUAL	MPIRT_Bcast_channel_is_equal	MPIRT::Bcast_Channel::Is_equal
MPIRT_BCAST_CHANNEL_ATTACH_ATTR	MPIRT_Bcast_channel_attach_attr	MPIRT::Bcast_Channel::Attach_Attr
MPIRT_BCAST_CHANNEL_RETRIEVE_ATTR	MPIRT_Bcast_channel_retrieve_attr	MPIRT::Bcast_Channel::Retrieve_Attr
MPIRT_BCAST_CHANNEL_DELETE_ATTR	MPIRT_Bcast_channel_delete_attr	MPIRT::Bcast_Channel::Delete_Attr

Gather Channel Specific Operations		
Language Independent Specification	C	C++
MPIRT_GATHER_CHANNEL_CREATE	MPIRT_Gather_channel_create	MPIRT::Gather_Channel
MPIRT_GATHER_CHANNEL_GET_RECVINBUFITER	MPIRT_Gather_channel_get_recvinbufiter	MPIRT::Gather_Channel::Get_Recvinbufiter
MPIRT_GATHER_CHANNEL_GET_RECVOUTBUFITER	MPIRT_Gather_channel_get_recvoutbufiter	MPIRT::Gather_Channel::Get_Recvoutbufiter
MPIRT_GATHER_CHANNEL_GET_ROOT	MPIRT_Gather_channel_get_root	MPIRT::Gather_Channel::Get_Root
MPIRT_GATHER_CHANNEL_SET_RECVINBUFITER	MPIRT_Gather_channel_set_recvinbufiter	MPIRT::Gather_Channel::Set_Recvinbufiter
MPIRT_GATHER_CHANNEL_SET_RECVOUTBUFITER	MPIRT_Gather_channel_set_recvoutbufiter	MPIRT::Gather_Channel::Set_Recvoutbufiter
MPIRT_GATHER_CHANNEL_SET_ROOT	MPIRT_Gather_channel_set_root	MPIRT::Gather_Channel::Set_Root
MPIRT_GATHER_CHANNEL_GET_INBUFITER	MPIRT_Gather_channel_get_inbufiter	MPIRT::Gather_Channel::Get_Inbufiter
MPIRT_GATHER_CHANNEL_GET_OUTBUFITER	MPIRT_Gather_channel_get_outbufiter	MPIRT::Gather_Channel::Get_Outbufiter
MPIRT_GATHER_CHANNEL_GET_GROUP	MPIRT_Gather_channel_get_group	MPIRT::Gather_Channel::Get_Group
MPIRT_GATHER_CHANNEL_GET_QOS	MPIRT_Gather_channel_get_qos	MPIRT::Gather_Channel::Get_Qos
MPIRT_GATHER_CHANNEL_SET_INBUFITER	MPIRT_Gather_channel_set_inbufiter	MPIRT::Gather_Channel::Set_Inbufiter
MPIRT_GATHER_CHANNEL_SET_OUTBUFITER	MPIRT_Gather_channel_set_outbufiter	MPIRT::Gather_Channel::Set_Outbufiter
MPIRT_GATHER_CHANNEL_SET_GROUP	MPIRT_Gather_channel_set_group	MPIRT::Gather_Channel::Set_Group
MPIRT_GATHER_CHANNEL_SET_QOS	MPIRT_Gather_channel_set_qos	MPIRT::Gather_Channel::Set_Qos
MPIRT_GATHER_CHANNEL_GET_NAME	MPIRT_Gather_channel_get_name	MPIRT::Gather_Channel::Get_Name
MPIRT_GATHER_CHANNEL_SET_NAME	MPIRT_Gather_channel_set_name	MPIRT::Gather_Channel::Set_Name
MPIRT_GATHER_CHANNEL_DUP	MPIRT_Gather_channel_dup	MPIRT::Gather_Channel MPIRT::Gather_Channel::Dup MPIRT::Gather_Channel::Operator=
MPIRT_GATHER_CHANNEL_FREE	MPIRT_Gather_channel_free	MPIRT::~Gather_Channel MPIRT::Gather_Channel::Free
MPIRT_GATHER_CHANNEL_IS_EQUAL	MPIRT_Gather_channel_is_equal	MPIRT::Gather_Channel::Is_equal
MPIRT_GATHER_CHANNEL_ATTACH_ATTR	MPIRT_Gather_channel_attach_attr	MPIRT::Gather_Channel::Attach_Attr
MPIRT_GATHER_CHANNEL_RETRIEVE_ATTR	MPIRT_Gather_channel_retrieve_attr	MPIRT::Gather_Channel::Retrieve_Attr
MPIRT_GATHER_CHANNEL_DELETE_ATTR	MPIRT_Gather_channel_delete_attr	MPIRT::Gather_Channel::Delete_Attr

Scatter Channel Specific Operations		
Language Independent Specification	C	C++
MPIRT_SCATTER_CHANNEL_CREATE	MPIRT_Scatter_channel_create	MPIRT::Scatter_Channel
MPIRT_SCATTER_CHANNEL_GET_RECVINBUFITER	MPIRT_Scatter_channel_get_recvinbufiter	MPIRT::Scatter_Channel::Get_Recvinbufiter
MPIRT_SCATTER_CHANNEL_GET_RECVOUTBUFITER	MPIRT_Scatter_channel_get_recvoutbufiter	MPIRT::Scatter_Channel::Get_Recvoutbufiter
MPIRT_SCATTER_CHANNEL_GET_ROOT	MPIRT_Scatter_channel_get_root	MPIRT::Scatter_Channel::Get_Root
MPIRT_SCATTER_CHANNEL_SET_RECVINBUFITER	MPIRT_Scatter_channel_set_recvinbufiter	MPIRT::Scatter_Channel::Set_Recvinbufiter
MPIRT_SCATTER_CHANNEL_SET_RECVOUTBUFITER	MPIRT_Scatter_channel_set_recvoutbufiter	MPIRT::Scatter_Channel::Set_Recvoutbufiter
MPIRT_SCATTER_CHANNEL_SET_ROOT	MPIRT_Scatter_channel_set_root	MPIRT::Scatter_Channel::Set_Root
MPIRT_SCATTER_CHANNEL_GET_INBUFITER	MPIRT_Scatter_channel_get_inbufiter	MPIRT::Scatter_Channel::Get_Inbufiter
MPIRT_SCATTER_CHANNEL_GET_OUTBUFITER	MPIRT_Scatter_channel_get_outbufiter	MPIRT::Scatter_Channel::Get_Outbufiter
MPIRT_SCATTER_CHANNEL_GET_GROUP	MPIRT_Scatter_channel_get_group	MPIRT::Scatter_Channel::Get_Group
MPIRT_SCATTER_CHANNEL_GET_QOS	MPIRT_Scatter_channel_get_qos	MPIRT::Scatter_Channel::Get_Qos
MPIRT_SCATTER_CHANNEL_SET_INBUFITER	MPIRT_Scatter_channel_set_inbufiter	MPIRT::Scatter_Channel::Set_Inbufiter
MPIRT_SCATTER_CHANNEL_SET_OUTBUFITER	MPIRT_Scatter_channel_set_outbufiter	MPIRT::Scatter_Channel::Set_Outbufiter
MPIRT_SCATTER_CHANNEL_SET_GROUP	MPIRT_Scatter_channel_set_group	MPIRT::Scatter_Channel::Set_Group
MPIRT_SCATTER_CHANNEL_SET_QOS	MPIRT_Scatter_channel_set_qos	MPIRT::Scatter_Channel::Set_Qos
MPIRT_SCATTER_CHANNEL_GET_NAME	MPIRT_Scatter_channel_get_name	MPIRT::Scatter_Channel::Get_Name
MPIRT_SCATTER_CHANNEL_SET_NAME	MPIRT_Scatter_channel_set_name	MPIRT::Scatter_Channel::Set_Name
MPIRT_SCATTER_CHANNEL_DUP	MPIRT_Scatter_channel_dup	MPIRT::Scatter_Channel MPIRT::Scatter_Channel::Dup MPIRT::Scatter_Channel::Operator=
MPIRT_SCATTER_CHANNEL_FREE	MPIRT_Scatter_channel_free	MPIRT::∼Scatter_Channel MPIRT::Scatter_Channel::Free
MPIRT_SCATTER_CHANNEL_IS_EQUAL	MPIRT_Scatter_channel_is_equal	MPIRT::Scatter_Channel::Is_equal
MPIRT_SCATTER_CHANNEL_ATTACH_ATTR	MPIRT_Scatter_channel_attach_attr	MPIRT::Scatter_Channel::Attach_Attr
MPIRT_SCATTER_CHANNEL_RETRIEVE_ATTR	MPIRT_Scatter_channel_retrieve_attr	MPIRT::Scatter_Channel::Retrieve_Attr
MPIRT_SCATTER_CHANNEL_DELETE_ATTR	MPIRT_Scatter_channel_delete_attr	MPIRT::Scatter_Channel::Delete_Attr

Reduce Channel Specific Operations		
Language Independent Specification	C	C++
MPIRT_REDUCE_CHANNEL_CREATE	MPIRT_Reduce_channel_create	MPIRT::Reduce_Channel
MPIRT_REDUCE_CHANNEL_GET_RECVINBUFITER	MPIRT_Reduce_channel_get_recvinbufiter	MPIRT::Reduce_Channel::Get_Recvinbufiter
MPIRT_REDUCE_CHANNEL_GET_RECVOUTBUFITER	MPIRT_Reduce_channel_get_recvoutbufiter	MPIRT::Reduce_Channel::Get_Recvoutbufiter
MPIRT_REDUCE_CHANNEL_GET_OP	MPIRT_Reduce_channel_get_op	MPIRT::Reduce_Channel::Get_Op
MPIRT_REDUCE_CHANNEL_GET_ROOT	MPIRT_Reduce_channel_get_root	MPIRT::Reduce_Channel::Get_Root
MPIRT_REDUCE_CHANNEL_SET_RECVINBUFITER	MPIRT_Reduce_channel_set_recvinbufiter	MPIRT::Reduce_Channel::Set_Recvinbufiter
MPIRT_REDUCE_CHANNEL_SET_RECVOUTBUFITER	MPIRT_Reduce_channel_set_recvoutbufiter	MPIRT::Reduce_Channel::Set_Recvoutbufiter
MPIRT_REDUCE_CHANNEL_SET_OP	MPIRT_Reduce_channel_set_op	MPIRT::Reduce_Channel::Set_Op
MPIRT_REDUCE_CHANNEL_SET_ROOT	MPIRT_Reduce_channel_set_root	MPIRT::Reduce_Channel::Set_Root
MPIRT_REDUCE_CHANNEL_GET_INBUFITER	MPIRT_Reduce_channel_get_inbufiter	MPIRT::Reduce_Channel::Get_Inbufiter
MPIRT_REDUCE_CHANNEL_GET_OUTBUFITER	MPIRT_Reduce_channel_get_outbufiter	MPIRT::Reduce_Channel::Get_Outbufiter
MPIRT_REDUCE_CHANNEL_GET_GROUP	MPIRT_Reduce_channel_get_group	MPIRT::Reduce_Channel::Get_Group
MPIRT_REDUCE_CHANNEL_GET_QOS	MPIRT_Reduce_channel_get_qos	MPIRT::Reduce_Channel::Get_Qos
MPIRT_REDUCE_CHANNEL_SET_INBUFITER	MPIRT_Reduce_channel_set_inbufiter	MPIRT::Reduce_Channel::Set_Inbufiter
MPIRT_REDUCE_CHANNEL_SET_OUTBUFITER	MPIRT_Reduce_channel_set_outbufiter	MPIRT::Reduce_Channel::Set_Outbufiter
MPIRT_REDUCE_CHANNEL_SET_GROUP	MPIRT_Reduce_channel_set_group	MPIRT::Reduce_Channel::Set_Group
MPIRT_REDUCE_CHANNEL_SET_QOS	MPIRT_Reduce_channel_set_qos	MPIRT::Reduce_Channel::Set_Qos
MPIRT_REDUCE_CHANNEL_GET_NAME	MPIRT_Reduce_channel_get_name	MPIRT::Reduce_Channel::Get_Name
MPIRT_REDUCE_CHANNEL_SET_NAME	MPIRT_Reduce_channel_set_name	MPIRT::Reduce_Channel::Set_Name
MPIRT_REDUCE_CHANNEL_DUP	MPIRT_Reduce_channel_dup	MPIRT::Reduce_Channel MPIRT::Reduce_Channel::Dup MPIRT::Reduce_Channel::Operator=
MPIRT_REDUCE_CHANNEL_FREE	MPIRT_Reduce_channel_free	MPIRT::∼Reduce_Channel MPIRT::Reduce_Channel::Free
MPIRT_REDUCE_CHANNEL_IS_EQUAL	MPIRT_Reduce_channel_is_equal	MPIRT::Reduce_Channel::Is_equal
MPIRT_REDUCE_CHANNEL_ATTACH_ATTR	MPIRT_Reduce_channel_attach_attr	MPIRT::Reduce_Channel::Attach_Attr
MPIRT_REDUCE_CHANNEL_RETRIEVE_ATTR	MPIRT_Reduce_channel_retrieve_attr	MPIRT::Reduce_Channel::Retrieve_Attr
MPIRT_REDUCE_CHANNEL_DELETE_ATTR	MPIRT_Reduce_channel_delete_attr	MPIRT::Reduce_Channel::Delete_Attr

Barrier Channel Specific Operations		
Language Independent Specification	C	C++
MPIRT_BARRIER_CHANNEL_CREATE	MPIRT_Barrier_channel_create	MPIRT::Barrier_Channel
MPIRT_BARRIER_CHANNEL_GET_INBUFITER	MPIRT_Barrier_channel_get_inbufiter	MPIRT::Barrier_Channel::Get_Inbufiter
MPIRT_BARRIER_CHANNEL_GET_OUTBUFITER	MPIRT_Barrier_channel_get_outbufiter	MPIRT::Barrier_Channel::Get_Outbufiter
MPIRT_BARRIER_CHANNEL_GET_GROUP	MPIRT_Barrier_channel_get_group	MPIRT::Barrier_Channel::Get_Group
MPIRT_BARRIER_CHANNEL_GET_QOS	MPIRT_Barrier_channel_get_qos	MPIRT::Barrier_Channel::Get_Qos
MPIRT_BARRIER_CHANNEL_SET_INBUFITER	MPIRT_Barrier_channel_set_inbufiter	MPIRT::Barrier_Channel::Set_Inbufiter
MPIRT_BARRIER_CHANNEL_SET_OUTBUFITER	MPIRT_Barrier_channel_set_outbufiter	MPIRT::Barrier_Channel::Set_Outbufiter
MPIRT_BARRIER_CHANNEL_SET_GROUP	MPIRT_Barrier_channel_set_group	MPIRT::Barrier_Channel::Set_Group
MPIRT_BARRIER_CHANNEL_SET_QOS	MPIRT_Barrier_channel_set_qos	MPIRT::Barrier_Channel::Set_Qos
MPIRT_BARRIER_CHANNEL_GET_NAME	MPIRT_Barrier_channel_get_name	MPIRT::Barrier_Channel::Get_Name
MPIRT_BARRIER_CHANNEL_SET_NAME	MPIRT_Barrier_channel_set_name	MPIRT::Barrier_Channel::Set_Name
MPIRT_BARRIER_CHANNEL_DUP	MPIRT_Barrier_channel_dup	MPIRT::Barrier_Channel MPIRT::Barrier_Channel::Dup MPIRT::Barrier_Channel::Operator=
MPIRT_BARRIER_CHANNEL_FREE	MPIRT_Barrier_channel_free	MPIRT::~Barrier_Channel MPIRT::Barrier_Channel::Free
MPIRT_BARRIER_CHANNEL_IS_EQUAL	MPIRT_Barrier_channel_is_equal	MPIRT::Barrier_Channel::Is_equal
MPIRT_BARRIER_CHANNEL_ATTACH_ATTR	MPIRT_Barrier_channel_attach_attr	MPIRT::Barrier_Channel::Attach_Attr
MPIRT_BARRIER_CHANNEL_RETRIEVE_ATTR	MPIRT_Barrier_channel_retrieve_attr	MPIRT::Barrier_Channel::Retrieve_Attr
MPIRT_BARRIER_CHANNEL_DELETE_ATTR	MPIRT_Barrier_channel_delete_attr	MPIRT::Barrier_Channel::Delete_Attr

Chapter 9

Channel Operations

Current Status: First formal vote passed 7/0/0 on 5/20/98.
Second formal vote passed 7/1/3 on 7/22/98.
Third formal voted passed 8/0/0 on 12/3/98.

⊤ (final)

This chapter describes the message transfer and related operations for channels. The chapter 6 describes the general operations for the channels, conditions and QoS for the channels. The objects of type MPIRT_PTCHANNEL, which are point-to-point channels and collective channels, all share these operations: MPIRT_CHANNEL_START, MPIRT_CHANNEL_ACTIVATE, MPIRT_CHANNEL_DEACTIVATE to start and to stop transfer of messages, MPIRT_CHANNEL_WAIT to determine the completion of a message transfer of a started channel, and MPIRT_CHANNEL_TEST to determine the status of the channel. ⊤ (final2)

All the operations of this chapter are defined for the committed channels. An attempt to use any of the operations defined in this chapter on an uncommitted channel results in MPIRT_ERR_OBJECT_NOT_COMMITTED error. ⊥ (final2)

9.1 Data Transfers

9.1.1 Performance Considerations

Several parameters given to the channel via the buffer iterator as described in Chapter 5.4 are aimed at allowing an implementation to optimize data transfers. Recall that these are allowed_buffer_list, policy, dataspec, and bufsize.

A buffer iterator can be shared among channels. All channels that share a buffer iterator are allowed to transfer data using any buffer that is in the buffer iterator's allowed_buffer_list. Also, all channels that share the buffer iterator will transfer the data specified by the same data descriptor defined by dataspec and bufsize given to the buffer iterator. However, users are allowed to define the QoS of the shared channels such that only a specific subset of the buffers in the allowed_buffer_list will ever be used by a specific channel that share the buffer iterator.

Advice to implementors.

QoS attached to the channels may determine that only a subset of the buffers in the allowed_buffer_list will ever be used for one of the given channels. The implementation is encouraged to optimize even further in this case, if possible.

In addition, an allowed_buffer_list also gives the implementation the information it needs to pre-plan data transfers at commit-time because only buffers from the allowed_buffer_list are allowed to be transferred over the channel at run-time. If an allowed_buffer_list is provided by the user, that is, the parameter is not given as MPIRT_CONTAINER_NULL, the scope of transfer optimizations is narrowed for the implementation. Only buffers that are in the allowed_buffer_list need be examined for possible optimization. If the allowed_buffer_list is specified as a *V-container* and any of the mapped or ordered buffer iterator policies, the implementation may make use of production and consumption order, endpoint buffer matching, and/or QoS promised or required by the user to further optimize transfers.

(*End of advice to implementors.*)

9.1.2 Channel States and Transitions

Figure 9.1 describes the state transition diagram for point-to-point and collective channels. The state transition diagram depicts the two-phase process of channel creation and shows the steps that channels go through when transferring a message.

The channel can be in one of the following states as defined in Figure 9.1:

UNCOMMITTED — The channel endpoint is created but has not been committed. It is not qualified for message transfer.

IDLE — The channel endpoint is committed, is qualified for message transfer, and is capable of a transition to MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED state for message transfer(s).

STARTED — The channel endpoint is committed, and able to transfer a message, provided the other participants in the transfer (channel other endpoints) are in either the MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED state. (It is erroneous to activate or deactivate a started channel.)

ACTIVATED — The channel endpoint is capable of multiple message transfers according to channel QoS, provided the other participants in the transfer (channel other endpoints) are in either the MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED state.

QOS_ERROR — The channel was not able to satisfy the channel's quality-of- service (QoS). Either a timeout expired, a stop event arrived for a timeout specified by the event, or a deadline has expired. Upon entering this state, the condition MPIRT_CHANNEL_COND_QOS_ERROR occurs and the event will be generated if there exist committed receptors for this event kind. For more details on a channel's QoS specification, see Chapter 11.

If there exists an implicit trigger for a channel condition (created by an implementation when a user creates an explicit receptor for the channel condition or when a user requests an implicit receptor for the channel condition by specifying this condition for this channel in an event name in some other channel quality of service), then an event is raised on this trigger (implicit trigger is raised) when the condition occurs.

It is an error to issue any of the operations defined in this chapter on a channel that is in the MPIRT_CHANNEL_UNCOMMITTED state.

A channel endpoint cannot be used for message transfers until it is committed. Upon creation, a channel endpoint is in the MPIRT_CHANNEL_UNCOMMITTED state and is transitioned into a MPIRT_CHANNEL_IDLE state when committed. A channel endpoint cannot transmit any messages while in the MPIRT_CHANNEL_IDLE state. MPIRT_CHANNEL_ACTIVATE not only transitions the local channel endpoint from MPIRT_CHANNEL_IDLE to MPIRT_CHANNEL_ACTIVATED state but is also disables the local channel endpoint from performing implicit message transfers. This transition has no impact on the states of remote channel endpoints. If any channel endpoint (collective channels can have multiple channel endpoints) is in the MPIRT_CHANNEL_IDLE state, no messages may be transferred over the channel, even if other channel endpoints are in MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED states.

The user is allowed to enable transfer of messages with channels by using one of two calls — MPIRT_CHANNEL_START or MPIRT_CHANNEL_ACTIVATE. If MPIRT_CHANNEL_START is called on a channel endpoint, that endpoint transitions into MPIRT_CHANNEL_STARTED state. MPIRT_CHANNEL_START enables exactly one message to be transferred over the channel according to the channel QoS. The MPIRT_CHANNEL_START call is non-blocking and the message transfer may not happen immediately if channel QoS specification requires it to be performed at some later time. If other channel endpoints are MPIRT_CHANNEL_IDLE, the message transfer will be delayed arbitrarily long until all channel endpoints of the channel have transitioned to either MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED state. After a single message has been transferred, the local endpoint returns from MPIRT_CHANNEL_STARTED to MPIRT_CHANNEL_IDLE state. It is an error to use MPIRT_CHANNEL_ACTIVATE or MPIRT_CHANNEL_DEACTIVATE on a channel endpoint that is in the MPIRT_CHANNEL_STARTED state.

MPIRT_CHANNEL_ACTIVATE enables messages to be transferred over the channel according to the channel QoS. If MPIRT_CHANNEL_ACTIVATE is called on a channel endpoint, that endpoint transitions into the MPIRT_CHANNEL_ACTIVATED state. Message transfers occur according to the channel's QoS specification and the state(s) of the channel other endpoint(s). When a message is transferred, the channel endpoint remains in the MPIRT_CHANNEL_ACTIVATED state and may continue to transfer messages. If all channel endpoints are in the MPIRT_CHANNEL_ACTIVATED state, messages will be transferred according to QoS without further application intervention.

When there is a QoS error on a channel, the buffers that should be involved in the message transfer are still moved from the input buffer iterator to the output buffer iterator for all channel endpoints, and it is the user's responsibility to deal with this condition by defining explicit receptor(s) for QoS error channel condition. After the implicit trigger for this condition is raised and all existing receptors are notified, the channel state then transfers to the MPIRT_CHANNEL_IDLE state if the channel was in MPIRT_CHANNEL_STARTED state or remains in the MPIRT_CHANNEL_ACTIVATED state if the channel was in MPIRT_CHANNEL_ACTIVATED state. QoS errors are not "recoverable" in the sense of the synchronous handler definition. The user is responsible for recovering from this condition.

The figure 9.1 describes the channel endpoint states and transitions between them. The solid circles and arrows represent states and transitions, respectively. The thick (double) arrows represent conditions that are generated by the channel. (If there are any receptors for the event kinds that represent these conditions then an implementation creates an implicit trigger for that channel event.) When a channel is created by one of the MPIRT_CHANNEL_CREATE type operations it is in the MPIRT_CHANNEL_UNCOMMITTED state. Analogously, an appropriate MPIRT_CHANNEL_FREE operation or MPIRT_OBJECT_FREE with the channel as its input parameter will destroy the channel. The MPIRT_COMMIT, if successful, commits the channel and transitions the channel into the MPIRT_CHANNEL_IDLE state. All other states of the channel are the states the channel is in while it is committed. When the channel is uncommitted by the MPIRT_UNCOMMIT operation or by the MPIRT_COMMIT operation when the channel is not in the designated container or one of its nested containers it transitions into the MPIRT_CHANNEL_UNCOMMITTED state. If the channel appears in the designated container or its nested containers then the MPIRT_COMMIT operation preserves the channel's MPIRT_CHANNEL_IDLE state.

When the channel is started (MPIRT_CHANNEL_START) then it transitions into the MPIRT_CHANNEL_STARTED state. Analogously when the channel is activated (MPIRT_CHANNEL_ACTIVATE), the channel transitions into the MPIRT_CHANNEL_ACTIVATED state. The transition into these two states does not mean that the message is being transferred. The transfer takes place according to the channel QoS and the states of channel other endpoints. That is depicted by an internal "substate" *Waiting for Transfer*. When the transfer takes place the channel first transitions into a *Transferring* substate within MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED state. Upon this transition the MPIRT_CHANNEL_COND_START (*start*) condition is generated (and an event with the same name for this channel endpoint if needed). If the transfer was successful then for the MPIRT_CHANNEL_STARTED channel it returns back to the MPIRT_CHANNEL_IDLE state and for the MPIRT_CHANNEL_ACTIVATED channel it remains in the MPIRT_CHANNEL_ACTIVATED state "waiting" for the next transfer. In either case the MPIRT_CHANNEL_COND_COMP (*completion*) condition is generated upon successful completion of the message transfer (and an event with the same name for this channel endpoint if needed). The return from the MPIRT_CHANNEL_STARTED to the MPIRT_CHANNEL_IDLE state is automatic and is done without user participation as depicted on the figure by a transition arrow with an arrow labeled "A" (not a transition with an operation name). A channel transitions back from the MPIRT_CHANNEL_ACTIVATED state upon user issued MPIRT_CHANNEL_DEACTIVATE operation.

The quality of service error can only occur when the channel is either transferring a message or when it is waiting to transfer a message. Hence, the channel is either in MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED state. When the quality of service error occurs the condition MPIRT_CHANNEL_COND_QOS_ERROR (*QoS Error*) is internally generated (and the event with the same name for this channel endpoint if needed) and the channel transitions into the MPIRT_CHANNEL_QOS_ERROR state. Recall that even in the case of the error the buffer is transferred from the channel endpoint input buffer iterator to the output buffer iterator (unless the error was caused by the channel input buffer iterator underflow or the channel output buffer iterator overflow). A local handler for the receptor for the implicit trigger for that condition can transition the channel "back" to the "operational" state. For the channel that was prior in the MPIRT_CHANNEL_STARTED state a handler may do nothing and that will transition the channel back to the MPIRT_CHANNEL_IDLE state (depicted by the transition arrow with the label "B"). This is the default behavior for the started channel in the absence of the local receptor for the channel QoS error. For the channel that was prior in the MPIRT_CHANNEL_ACTIVATED state a handler may do nothing and that will transition the channel back to the MPIRT_CHANNEL_ACTIVATED state (in the *Waiting for Transfer* substate). This is depicted by the transition arrow with label "C". This is the default behavior for the activated channel in the absence of the local receptor for the channel QoS error. A handler may issue MPIRT_CHANNEL_DEACTIVATE the channel which will transition it into the MPIRT_CHANNEL_IDLE state (depicted by the transition labeled "deactivated").

9.1.3 Methods for Single Message Transfer

A message transfer method specified in this section starts the transfer of one and only one message from/to a single buffer of the channel input buffer iterator over one point-to-point channel or one collective channel. As such, it is possible to use the mechanisms defined here to perform flow control at the granularity of a single message.

> *Advice to users.* It is legal to have multiple channels drawing from one buffer iterator. Indeed, this may be advantageous in the case of fault-tolerant systems, where two channels can be attached to the same buffer iterator. When a buffer is placed into the buffer iterator, only one channel will transfer the data in the buffer. (*End of advice to users.*)

To transfer data that is in a buffer, first insert the buffer into the input buffer iterator of the desired channel, and then use MPIRT_CHANNEL_START to initiate communication for that message on the channel. The actual start of the message transfer is done according to the QoS of the channel (point-to-point, or collective). If the QoS is MPIRT_QOS_BEST_EFFORT, then the start operation sends the message whenever feasible, which may be immediately. The channel will transfer the first buffer from the input buffer iterator as defined by the buffer iterator policy.

MPIRT_CHANNEL_START (channel)

INOUT	channel	point-to-point or collective channel (ref. to MPIRT_CHANNEL)

```
int MPIRT_Channel_Start (MPIRT_Channel *channel)
```

```
int MPIRT::Channel::Start (void)
```

It is an error to start a channel endpoint that is in the MPIRT_CHANNEL_STARTED or in the MPIRT_CHANNEL_ACTIVATED state.

The effect of the MPIRT_CHANNEL_START operation on a channel is to first transition the channel endpoint into the MPIRT_CHANNEL_STARTED state and then remove a single buffer from the input buffer iterator and transfer its message over that channel to the destination. Either end of the channel can use MPIRT_CHANNEL_START to start a message transfer. This allows MPI/RT to support both **push** and **pull** models of one-sided communication. Upon completion of the message transfer the buffer that contained the message is inserted into the output buffer iterator of the channel, and finally the channel transition back into the MPIRT_CHANNEL_IDLE state. The user can determine the completion of a message transfer operation by using MPIRT_CHANNEL_WAIT or MPIRT_CHANNEL_TEST operations or by using explicit receptors with handlers for the channel events.

9.1.4 Methods for Multiple Message Transfers

For high-performance systems, it is important to enable the system to transfer or receive messages as soon as the network will allow it or as soon as a message buffer is ready. In order to support this, the MPIRT_CHANNEL_ACTIVATE operation activates data transfers over a channel according to the channel's QoS. Using the event-driven paradigm and specifying appropriate events, the message transfer can take place as soon as a buffer with the message is inserted into the input buffer iterator of a sending endpoint of a channel and a free buffer is available in the input buffer iterator of a receive endpoint of a channel. Thereafter, users only need to use combinations of MPIRT_BUFITER_INSERT and MPIRT_BUFITER_REMOVE on channel endpoints to move data over the channel.

Either both endpoints of a point-to-point channel or all endpoints of a collective channel must call MPIRT_CHANNEL_ACTIVATE to enter the MPIRT_CHANNEL_ACTIVATED state that supports a zero-sided communication mode. Once in the MPIRT_CHANNEL_ACTIVATED state, a channel endpoint remains in that state until the MPIRT_CHANNEL_DEACTIVATE operation is called on it. Calling

MPIRT_CHANNEL_ACTIVATE on only one end of a channel opens it for message transfer control by the other endpoint(s) of the channel by calling MPIRT_CHANNEL_START on the other endpoint(s).

The following function activates the channel endpoint for message transfers.

MPIRT_CHANNEL_ACTIVATE (channel)

INOUT	channel	point-to-point or collective channel (ref. to MPIRT_CHANNEL)

`int MPIRT_Channel_activate (MPIRT_Channel *channel)`

`int MPIRT::Channel::Activate (void)`

⊤ (final2) It is an error to call MPIRT_CHANNEL_ACTIVATE on a channel endpoint that is in the MPIRT_CHANNEL_STARTED or in the MPIRT_CHANNEL_ACTIVATED state. ⊥ (final2)

The following function disables an activated channel endpoint.

MPIRT_CHANNEL_DEACTIVATE (channel)

INOUT	channel	point-to-point or collective channel (ref. to MPIRT_CHANNEL)

`int MPIRT_Channel_deactivate (MPIRT_Channel *channel)`

`int MPIRT::Channel::Deactivate (void)`

A call to MPIRT_CHANNEL_DEACTIVATE must complete all in-progress message transfers over ⊤ (final2) the channel before transitioning the channel endpoint into the MPIRT_CHANNEL_IDLE state and returning control to the user program.

It is an error to call this function on a channel endpoint that is in the MPIRT_CHANNEL_STARTED state. It is not an error to deactivate a channel endpoint that is in the MPIRT_CHANNEL_IDLE state. ⊥ (final2)

9.2 Operations for Testing Completion and Determining the State of Data Transfers

The channels support wait and test operations and allow the user to monitor the progress of a channel message transfer.

> *Advice to users.* While this standard does not define a specific functionality to extend the status monitoring to any user activity, users can use receptors and handlers for specific event kinds to determine the completion of any activity of interest. The activity of interest will create an event trigger with some predefined name, and any other interested user activities will create event receptors with required handlers for that event type.
>
> More elaborate wait functionalities, like waiting for multiple channel transfer completions or waiting on transfer completion on multiple channels, can be achieved by creating a common receptor for multiple channel events. The handler for that event receptor will notify, or wake up, a user activity when the appropriate state among these channels is reached. (*End of advice to users.*)

9.2.1 Wait Operation

The following operation allows users to wait on the completion of a channel message transfer. This call is blocking.

MPIRT_CHANNEL_WAIT (channel, flag)

IN	channel	point-to-point or collective channel (MPIRT_CHANNEL)
OUT	flag	completion status for the channel message transfer (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Channel_Wait (MPIRT_Channel channel, MPIRT_Boolean *flag)
```

```
int MPIRT::Channel::Wait (MPIRT::Boolean &flag) const
```

The flag indicates whether the message transfer over the channel was successful or not. The value MPIRT_TRUE indicates that the message transfer was successful, while the value MPIRT_FALSE indicates that the message transfer was not successful. Notice that the return value of the operation MPIRT_CHANNEL_WAIT indicates the success or failure of the wait operation and not the message transfer operation.

⊤ (final2) It is an error to wait on a channel endpoint that is in the MPIRT_CHANNEL_UNCOMMITTED state. It is not an error to wait on a channel endpoint that is in the MPIRT_CHANNEL_IDLE state. If a channel is in the MPIRT_CHANNEL_IDLE state then MPIRT_CHANNEL_WAIT returns immediately with status MPIRT_SUCCESS and flag equal to MPIRT_TRUE. If a channel is in the MPIRT_CHANNEL_ACTIVATED state, it is illegal to issue MPIRT_CHANNEL_WAIT operation. The return status for MPIRT_CHANNEL_WAIT in this case is MPIRT_ERR_ACTIVE_CHANNEL. ⊥ (final2)

9.2.2 Test Operation

The test operation returns the state of the committed channel endpoint at the time it is called. This function returns immediately and is non-blocking.

MPIRT_CHANNEL_TEST (channel, status)

IN	channel	point-to-point or collective channel (MPIRT_CHANNEL)
OUT	status	state of the channel (ref. to MPIRT_CHANNEL_STATUS)

```
int MPIRT_Channel_Test (MPIRT_Channel channel, MPIRT_Channel_status *status)
```

```
int MPIRT::Channel::Test (MPIRT::Channel::Status &status) const
```

The status returns the state of the channel at that moment.

⊤ (final2) It is an error to test a channel endpoint that is in the MPIRT_CHANNEL_UNCOMMITTED state. ⊥ (final2)

Advice to users. Users may not need to know the state of the committed channel or the completion of a channel message transfer at all if they are using time-driven or event-driven models. These two models define when the message transfer is finished via time instance or by scheduling other user activities on the events that are triggered by the channel message transfer completion. (*End of advice to users.*)

9.3 Summary

⊤ (final2) This section summarizes enumerations and type declarations that are introduced in this chapter.

Channel States		
Item	*What is it*	*Defined as*
MPIRT_CHANNEL_UNCOMMITTED	enumeration	channel endpoint has not been committed
MPIRT_CHANNEL_IDLE	enumeration	channel endpoint is idle (deactivated)
MPIRT_CHANNEL_ACTIVATED	enumeration	channel endpoint is activated
MPIRT_CHANNEL_STARTED	enumeration	channel endpoint is ready to perform data transfer or is performing a data transfer
MPIRT_CHANNEL_QOS_ERROR	enumeration	a QOS error has occurred

⊥ (final2)

General Channel Operations		
Language Independent Specification	C	C++
MPIRT_CHANNEL_START	`MPIRT_Channel_start`	`MPIRT::Channel::Start`
MPIRT_CHANNEL_ACTIVATE	`MPIRT_Channel_activate`	`MPIRT::Channel::Activate`
MPIRT_CHANNEL_DEACTIVATE	`MPIRT_Channel_deactivate`	`MPIRT::Channel::Deactivate`
MPIRT_CHANNEL_WAIT	`MPIRT_Channel_wait`	`MPIRT::Channel::Wait`
MPIRT_CHANNEL_TEST	`MPIRT_Channel_test`	`MPIRT::Channel::Test`

⊤ (final2)

Miscellaneous Bindings		
Language Independent Specification	*C Bindings*	*C++ Bindings*
MPIRT_CHANNEL_UNCOMMITTED	MPIRT_CHANNEL_UNCOMMITTED	MPIRT::Channel::UNCOMMITTED
MPIRT_CHANNEL_UNCOMMITTED	MPIRT_CHANNEL_UNCOMMITTED	MPIRT::Channel::UNCOMMITTED
MPIRT_CHANNEL_IDLE	MPIRT_CHANNEL_IDLE	MPIRT::Channel::IDLE
MPIRT_CHANNEL_ACTIVATED	MPIRT_CHANNEL_ACTIVATED	MPIRT::Channel::ACTIVATED
MPIRT_CHANNEL_STARTED	MPIRT_CHANNEL_STARTED	MPIRT::Channel::STARTED
MPIRT_CHANNEL_QOS_ERROR	MPIRT_CHANNEL_QOS_ERROR	MPIRT::Channel::QOS_ERROR

⊥ (final2)
⊥ (final)

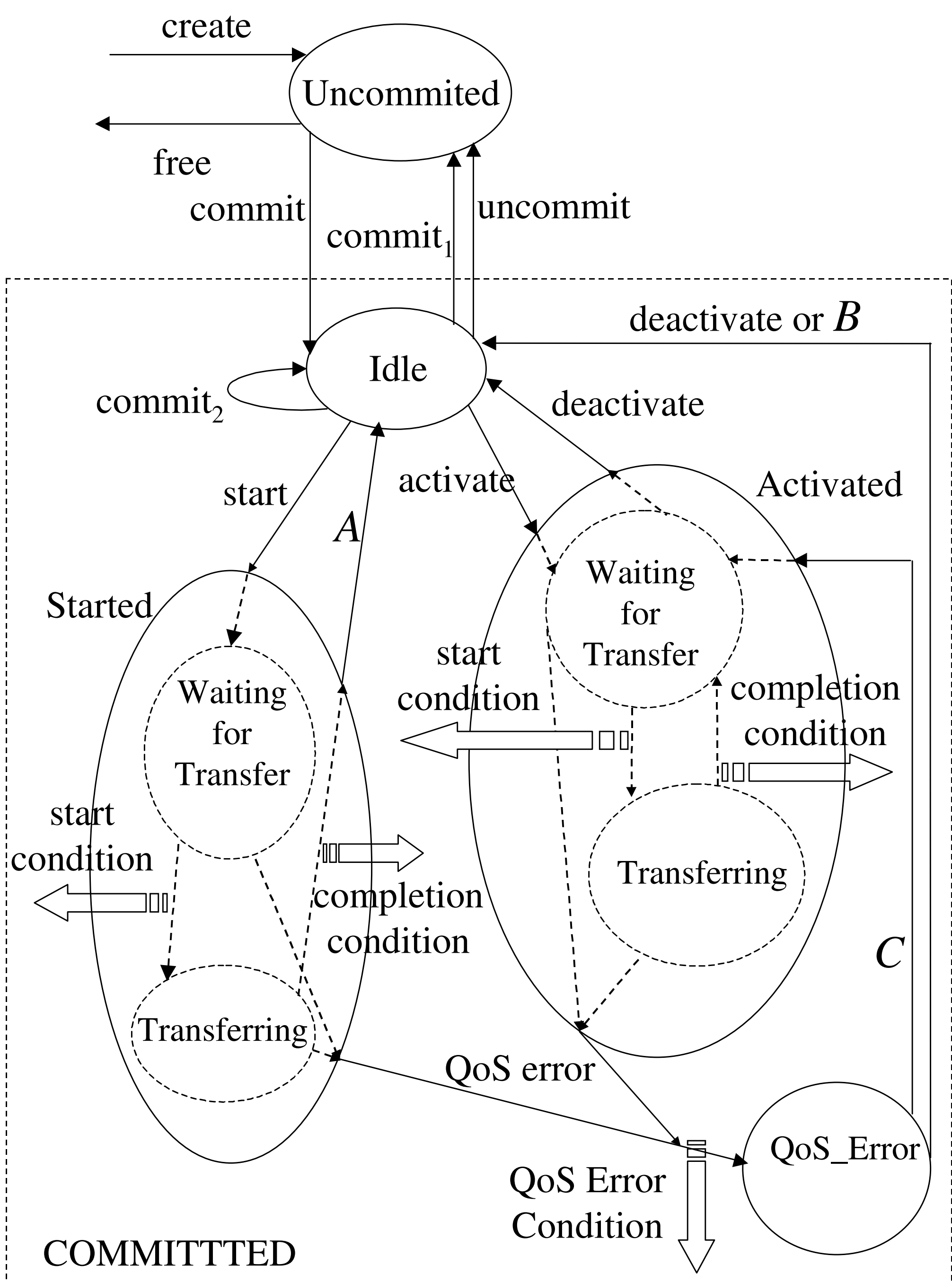

Figure 9.1: Channel state transition diagram.

Part III

Real-Time Programming Models and Qualities-of-Service

Chapter 10

Quality-of-Service Overview

Current Status: First formal vote on Section 10.1 passed 9/0/1 in February 98.
First formal vote on subsections 10.2.1 and 10.2.2 passed 10/0/0 in February 98.
First formal vote on subsection 11.2.1 passed 9/0/0 in May 98.
First formal vote on section 10.3 passed 7/0/2 in May 98.
First formal vote on section 10.4 passed 9/0/0 in May 98.
Second formal vote passed 6/0/3 in July 98.
Third formal vote passed 8/0/0 on 12/3/98.

10.1 Introduction

The quality-of-service (QoS) parameters are the most important distinction between other message-passing specifications[1] and MPI/RT. The QoS specification follows the same principles as the rest of MPI/RT, namely early deferred binding. QoS parameters are defined for MPI/RT objects rather than operations. This allows MPI/RT to guarantee QoS and provide better resource utilization by allowing implementations to perform early admission and/or resource reservation prior to any data transfer or other runtime operations. This approach contrasts other approaches where QoS is part of the late binding operation that increases implementation runtime overhead, makes global optimization impossible, and puts a higher burden on the application writers to cope with denial of service at a runtime vs. initialization.

The QoS specification consists of two parts: *QoS_constraint(s)* and the *QoS_requirement*. The first specifies the constraints for start and/or finish of an operation on the object and uses the format of events and timeouts. The second specifies the QoS_requirement that the user requested for an operation on the object and uses the format of priority or time instances that represent the start and the deadline. This standard defines QoS specifications for channels (both point-to-point channels and collective channels), handlers, and event delivery abstractions (both event triggers and event receptors).

The QoS for a channel specifies QoS_constraints like starting and finishing events and timeouts, QoS_requirements like deadline and priority for a message transmission, and the starting time for a message transmission. The QoS for a handler specifies a priority as the QoS_requirement and a timeout as the finish QoS_constraint. For event delivery abstraction, a separate specification is defined for the QoS for event triggers and event receptors. For an event receptor, the QoS specifies

[1] including MPI.

the bound on the rate of the event generation as a QoS_constraint. Finally, for event receptors, QoS specifies either a deadline as both the QoS_requirement and QoS_constraint or a priority as the QoS_requirement and a timeout as the finishing QoS_constraint for the delivery of an event from an event trigger to the receptor for which this QoS is specified. The QoS_constraint entities for the invocation and for the timeout for operations on objects that have QoS can be specified by a time instance for timeout or an implicit event receptor. The QoS_requirement for these objects can be specified by a time interval or a priority. While the QoS parameters specify application requests for these objects, the QoS guarantees that MPI/RT provides to the user affect the scheduling of multiple, and potentially different, resources. For example, the QoS specification for a point-to-point channel requests a guarantee for a message transfer that affects network resources; the QoS specification for a handler requests a guarantee for the handler execution that requires the operating system cooperation; and the QoS specification for an event receptor requests a guarantee for the event delivery that affects both network and operating system resources.

Only paradigm-specific parameters are part of the MPI/RT QoS specification for a channel. Any lower-level QoS parameters are only discussed as advice to implementors. See also [4, 13].

Below, QoS parameters for time-driven, event-driven, and priority-driven real-time paradigms are defined. The QoS specification that MPI/RT uses does not directly align along these paradigms, but rather combines some of them together. The individual "pure" real-time paradigms can still be specified by using MPIRT_QOS_*object-type_name-of-parameter*_IGNORE and MPIRT_QOS_*object-type_name-of-parameter*_NULL for the appropriate parts of the QoS specification.

The time-driven paradigm specifies QoS_constraint entities (time instances) and QoS_requirements (starting time and deadline). QoS_constraint entities define the start and the timeout of an activity, for example, message transfer (channel use), handler execution, and event propagation. The event-driven paradigm does not specify any QoS_requirement parameters; it only specifies the QoS_constraint entities (event receptors). The priority-driven paradigm in its pure form specifies only QoS_requirements (integer priority) but not QoS_constraint entities.

While the priority does not have any QoS_constraint entities explicitly specified, users can invoke a message transfer operation explicitly when a user defined timer expires. The other way to specify QoS_constraint entities with priority QoS_requirements is to use the combination of priority and event-driven paradigms as described below.

While the event-driven paradigm specifies QoS_constraint entities but does not specify QoS_requirements, one can use the combination of time-driven with event-driven paradigms or event-driven with priority paradigms. As described in the event-driven chapter (see Section 10.3), the time instance can be interpreted as a QoS_constraint that starts or stops an operation on the object for which the QoS is defined. Events will specify the QoS_constraint entities, while the deadline will specify the QoS_requirement. For the combination of the priority with the event-driven paradigms, events will specify QoS_constraint entities, and the priority will specify the QoS_requirement. Notice the separation of roles for QoS between the event triggers and the event receptors of the event delivery abstraction. An event receptor is a user specification of a QoS_constraint (guard) for starting or stopping an activity. So it is a notification mechanism from MPI/RT to the user. An event trigger is a notification mechanism to MPI/RT to let it know that some "event" of interest to the application or the implementation has happened. Handling the match between the event triggers and the event receptors is the role of the event delivery abstraction, and delivered events drive the event-driven model.

The next three sections of this chapter contain descriptions of real-time paradigms. The last section of this chapter contains the specification parameters for QoS.

10.2 Time-Driven Real-Time Programming Model

The primary goal of the time-driven approach to MPI/RT is to allow the real-time application sufficient control over the environment in which it is running so that it can explicitly schedule its message-passing activities and resource use. Since MPI/RT is designed as a message-passing library, it cannot schedule by itself, but must depend upon the operating system and communication and network protocols to enforce specified schedules.

An application using time-driven MPI/RT will be able to specify time intervals to bound the resource use of communication operations using globally synchronized clock values.

10.2.1 Scheduling Message Transfers

Existing message transfer operations[2] lack two parameters that are considered critical for real-time applications, especially for the time-driven programming paradigm. These are a starting time of the operation and a timeout or a deadline for completion of the operation. The starting time of an operation should be considered a special case of an event that specifies the requirement for a message transfer. Generally, an event is just a constraint that starts a message transfer, but the starting time of the time-driven paradigm is both the QoS_requirement and the QoS_constraint that the message transfer will not start before the starting time. While certain applications (especially embedded ones) prefer an even finer granularity of control, a balance between the feasibility of an implementation and what time-driven application designers want to use must be struck. For example, there is a hard lower bound for the starting time, but no hard upper bound on the starting time.

One distinctive characteristic of the time-driven approach to real-time message-passing is its lack of need for system queues and system buffers[3]. On many systems, this allows the removal of a ⊤ (final2)
hand-shake operation and results in improved performance. Since a parallel time-driven program ⊥ (final2)
must globally schedule all message transmissions, the message receiver always knows to expect an incoming message. Thus, for reasons of efficiency and simplicity, a time-driven MPI/RT implementation should not do any handshaking (as many of the existing non-real-time implementations do). It is up to the application to specify times (for start and timeout) to ensure that the sender/receiver (local/remote) pairs are working in synchrony.

Another distinctive feature is a potentially more efficient method of notification use, which can be more minimal (shorter critical instruction path) than with other approaches. A time-driven MPI/RT application does not need to be notified when a message is transmitted successfully and on time; instead it is notified only when an error occurs (e.g., a timeout expires). A matter of significant discussion in the MPI/RT group concerns precisely what should happen to messages left on the network when a timeout expires.

10.2.2 Schedulable Time Intervals

⊤ (final)

An activity interval, specified by a starting time and a deadline[4], is an input parameter for a ⊥ (final)
scheduled message transfer over a channel. The purpose of this parameter is to ensure that the

---

[2]See MPI.

[3]Applications use MPI *ready mode* message-passing implicitly. A ready-mode send may be started *only* if the matching receive has already been posted [5].

[4]A timeout does not require a guarantee from MPI/RT to complete an operation before timeout, while a deadline not only specifies timeout as a constraint but also requests a requirement guarantee from MPI/RT that an operation will complete before the deadline. In both cases, if an operation does not complete by the time specified by the timeout or the deadline, the operation returns with failure, and users can define handlers to be invoked on it.

system resources required to satisfy this operation will not be used outside of the specified interval. These resources can be narrowly interpreted to refer to the interprocess communications network. A broader interpretation would include memory accesses, node busses, network interface cards, and so on. Again, while a finer granularity of control is preferred, a balance between the feasibility of an implementation and what time-driven schedule designers want to use is desired.

The starting time and deadline are somewhat symmetric. The starting time ensures that the resources needed for a data transfer operation will be available at the specified start time. The deadline parameter, in contrast, would ideally specify the time when all resources required by the message transfer operation are free to be used by others. That is, after the time specified in the deadline, regardless of whether the operation completed successfully or not, all system resources (physical network, network interface cards, node buses, message buffers, etc.) have been released and can be used for subsequent message-passing operations.

> *Rationale.* Unfortunately, in practice these guarantees often cannot be met. The MPI/RT deadline therefore specifies that the message transfer should be stopped, the calling application should be notified if the operation has not completed by the time specified by the deadline, and all the resources are released as soon as possible. Since the message may be progressing through a multi-stage network, a time-driven MPI/RT implementation may need to send a message from the receiver node to the sender to indicate that the timeout has occurred. The resulting error messages may not be received by the deadline, and they may use resources after the deadline. Thus the application may need to reserve resources to handle such events. It may be extremely difficult to establish such bounds, especially if the real-time performance characteristics of the operating system or the underlying runtime system are poorly known or highly variable. (*End of rationale.*)

10.2.3 The MPI/RT Time Specification

The starting time and the deadline of an activity interval in time-driven MPI/RT message transfer operation calls are specified by the type MPIRT_TIME_SPEC (see Chapter 3). When the starting time is specified as the constant MPIRT_TIME_SPEC_IGNORE of type MPIRT_TIME_SPEC, then there is no hard requirement requested on when the operation should start. Implicitly, it should start as soon as possible[5]. Similarly, when a deadline is given by a constant MPIRT_TIME_SPEC_IGNORE, there will be no hard requirement on when the operation should end.

The activity interval for time-driven MPI/RT message transfers is specified by two parameters: the starting time and the deadline, each specified in turn by an instance of the type MPIRT_TIME_SPEC.

10.3 Event-Driven Real-Time Programming Model

The event-driven paradigm provides a mechanism for scheduling any application activity using system, application, and MPI/RT events as constraint-guards that control the start and the duration of the activity. This paradigm allows users to manage system and user resources. Thus, MPI/RT provides the interface for control flow by using events.

> *Advice to implementors.* An implementation can use this mechanism to manage its resources and activities. (*End of advice to implementors.*)

[5] This is the behavior of current MPI calls.

10.3.1 Overview

The main goal of the event-driven paradigm of MPI/RT is to help the application to control the run-time environment in which it is running, with scheduling of communication and computation activities and their resource use. Coordination is required between MPI/RT, the operating system, and communication and network protocols to enforce the schedules.

In summary, an application using event-driven MPI/RT will be able to specify intervals *guarded* by specified kinds of events in order to bound (constrain) the resource use of communication and computation activities.

> *Rationale.* Currently many applications "wait" on system events or user control messages to schedule a handler, which, in turn, may schedule several application activities: functions, processes, threads, and message transfers. The model for the event-driven paradigm presented in this document establishes the direct coupling between event kinds and application activities without user-created handlers and without a restriction that events must be local to the process. (*End of rationale.*)

Each kind of event is identified by its name (see Chapter 4). Events are not necessarily local to the process or even a node. The event kind definition identifies the scope of the delivery of events of that kind. Within the scope the event kind name is unique.

Event triggers and receptors define the scope of the event kind and the name of an event kind they generate and waiting for, respectively. MPI/RT delivers the events from the triggers to the receptors that share the event kind name and are within the scope of that event kind. MPI/RT also starts or aborts an object operation that is guarded (constrained) by the delivered events.

The event-driven real-time communication model follows the receiver-based communication paradigm for control flow. It consists of three parts.

1. A user requests that an implementation create implicit event receptors that guard the scheduling interval for the message transfer operations on a channel (i.e. message transfer over a channel). Guards for the scheduling interval that are QoS constraints are event receptors that start and stop an MPI/RT operation on a guarded channel. An event receptor contains an event name that is matched by MPI/RT to the event trigger (implicit or explicit). A user may not know where or how an event is raised. Users can define for the implicit event receptor a QoS as a requirement (deadline or priority) and a timeout for the delivery of event triggers to the event receptors as a constraint (see Section 11.4.2). For operations on event receptors, see Section 10.3.3.

2. A user either creates an explicit trigger or requests an implementation to create an implicit event trigger that raises the event, and MPI/RT in turn delivers them to the above created implicit event receptor. Again, the user who deals with triggers is not aware of where these events will be delivered or how these events are used by the event receptor if there are any receptors for the event trigger(s). Real-time paradigm users can request the QoS for an event trigger (see Section 11.4.1). For operations on event triggers, see Section 4.2.3.

3. A user "registers" the events with MPI/RT as part of the event delivery abstraction. This registration is done as part of MPIRT_COMMIT, and both triggers and receptors of the event delivery abstraction are the committable objects (see Chapter 12). This operation also returns an error if there is an event receptor without a corresponding event trigger.

The main purpose of the event receptors is to *guard* the interval when an object operation may use resources. This is analogous to the real-time time-driven paradigm where no resources will be used by a message transfer operation prior to its starting time of the channel QoS time interval and, to the best of the MPI/RT implementation effort, no resources will be used after the deadline of the channel QoS time interval.

The time interval of the real-time time-driven communication paradigm in essence contains two receptors that are specified by the time instances. From this perspective, the real-time time-driven communication paradigm is just a subset of the event-driven one. There are, however, two critical differences that lie in the ability of the application to schedule its non-MPI/RT activities. For the time-driven paradigm, there are existing facilities to start non-MPI/RT activities using OS timers, spin-locks, and others. These facilities and the synchronized clocks allow the application to coordinate all of its activities, both communication and non-communication ones, both local and global. There are no analogous mechanisms for the event-driven paradigm, and event delivery/monitoring across the entire platform requires application action and sufficient communication support. This is the place where MPI/RT can really help. The second big difference lies in the paradigms' interpretations of the events themselves. The event-driven paradigm views events as constraints only, while time-driven paradigm, in addition to the constraints, views the deadline and the start time as a QoS requirements that MPI/RT needs to satisfy.

10.3.2 Event Triggers

There are three different classes of events based on event triggerer classification: application, system and communication. Three issues need to be addressed for each event triggerer class.

1. Who generates an event?
2. Who is aware of the event?
3. What is the scope of the event?

System Events The type of the event trigger indicates the source of the event and consequently what resources are involved. System events are generated by the platform environment, for example the operating system, and use implicit triggers. A user is unaware of the triggers for these events but would like to be able to schedule their activity based upon events they raise. For example, an application or implementation can use these events for fault-tolerance. An MPI/RT implementation can detect these events and create implicit receptors.

> *Advice to users.* The MPI/RT standard does not define any system events. MPI/RT may define system events for fault-handling or instrumentation. (*End of advice to users.*)

> *Advice to implementors.* Implementors are free to use this mechanism to deliver platform-specific events for their users and provide documentation for them. (*End of advice to implementors.*)

The scope of system events is local to the node where the process that uses the event is running. Hence, the scope for these events is MPIRT_GROUP_SELF.

> *Advice to users.* MPI/RT may define system events scoped to a platform. (*End of advice to users.*)

Communication Events Currently all communication events are associated with MPI/RT channels. MPI/RT generates all communication events and consequently can detect them. MPI/RT contains five communication events associated with channel objects (see Chapter 6): MPIRT_CHANNEL_COND_COMP, MPIRT_CHANNEL_COND_START, and MPIRT_CHANNEL_COND_QOS_ERROR.

The names for the communication events are defined over scoped to the group over which the channel is defined, while the scope for the delivery of these events is defined over the group MPIRT_GROUP_WORLD. Thus, the event name should be unique over the scope of the group MPIRT_GROUP_WORLD. Users do not define the event name for channel events. Instead, a convention for the event name generation based on the name of the channel is used. Event names for each channel condition are named using the following system: {*channel name*}:{*rank*}:{*channel event*}. This naming convention uniquely identifies an event within the scope of the group over which the channel is defined. The rank represents the rank of the channel endpoint where the condition is raised. For example, if a user is interested in the completion of a data transfer on the sender side of a point-to-point channel named *channel#1* between the sender rank 1 and the receiver rank 3 over the group *my_group*, the application can create a receptor in any process with the event name channel#1:1:COMP.

Application Events An application event has meaning only to the application. MPI/RT is just a mechanism to match application event triggers and event receptors as well as the mechanism for event delivery from triggers to receptors. An application assigns a persistent name to an event kind and defines the scope of the event delivery. This is the only event class whose triggers are created by the users.

10.3.3 Event Receptors

Receptors receive events and either schedule handlers that a user defined for the receptor (user-created receptors) or start/stop a channel data transfer that they guard (user-requested receptors via QoS specification). An application can specify handlers to be launched on the event arrival for the receptor to start or stop any of its activities. This second type of (implicit) receptor "guards" a liveliness interval within which an MPI/RT object operation can use resources. Only objects of the type MPIRT_CHANNEL can be guarded by event receptors. The channel event arrival triggers a single message transfer over the channel which is guarded by this event receptor.

10.4 Priority-Driven Real-Time Programming Model

There is no separate functionality for the priority-driven paradigm.

In the priority-based real-time programming paradigm, process and message priorities are used to meet timing specifications. However, because process priorities can be handled in a variety of ways, it is extremely difficult to provide portable mechanisms to specify them, and MPI/RT does not directly address this issue.

Advice to implementors. There are several QoS parameters that are related to the priority-driven paradigm but are not part of the MPI/RT specification:

- Preemptability flag (a boolean value).
- Some measure of the preemption quantum desired in bytes (that allows a user to calculate timing bounds for guaranteed delivery) or a duration of time interval (like 10ms that many schedulers use).

Implementors can determine these parameters and use them for the QoS guarantees.

Channel preemption can be specified in either message length or the time interval of scheduling protocol stack activities. For the handler, only the scheduling time interval makes sense. In either specification, it is unlikely that the MPI/RT implementation can change these parameters from one object to another. Usually, these parameters are platform-wide. Thus, these QoS parameters should be specified once, either at the MPI/RT initialization time, or MPI/RT run command parameter, or system boot time. (*End of advice to implementors.*)

Priority specifies the QoS_requirement part of the QoS, but not any constraint entities. Starting and stopping constraints can be done explicitly by an application or an implementation. For that reason, MPI/RT decided to combine a priority real-time communication specification that defines the QoS_requirement with an event-driven real-time communication specification that defines QoS_constraint entities of QoS. The handler and event QoSs do not specify any constraint mechanism since the starting constraint is an explicit part of handler specification. However, it is common to have a stopping constraint. For this reason, a *timeout* is allowed to be specified together with the QoS. A timeout should not be confused with a *deadline*. While a timeout just defines the stop trigger, a deadline is used for scheduling of an activity (delivery of event, handler execution, or message transfer) and as a stop trigger. A timeout is usually defined with respect to the start of the activity, while a deadline can be defined with respect to the start of the activity or in absolute time form.

10.4.1 Channel Priority

In MPI/RT, priorities are specified for channels (point-to-point and collective), for handlers, and for the event delivery using a QoS parameter for object creation.

Because various platforms may provide different levels of support for message priority at the OS level and below, MPI/RT specifies very little about how message priorities are implemented. In addition to passing message priority information to the appropriate OS and hardware layers, a high-quality MPI/RT implementation will order operations internally according to priority information. For example, given the choice between performing two different communication operations (such as receiving one message or another), the higher priority communication should be performed first. If the high priority communication blocks or stalls, the lower priority communication may be initiated. Notice that in the general case, this implies that communication may need to be preempted. For example, if the user initiates a low-priority nonblocking send, then begins a high-priority send, the low-priority send would be stalled in favor of the high-priority send.

The handler's priority should be used by MPI/RT to schedule handlers or pass this information to the operating system to schedule the handler execution. Notice that priorities for the handlers define relative order between them, and MPI/RT must map that information to OS-provided priorities. A high-quality implementation may even map different handler priorities to different interrupt handlers.

For event delivery, the priority should be used for both local scheduling of MPI/RT event handling-related activities and for transferring an event over the communicator if this is necessary.

10.4.2 Process Priority

As stated above, MPI/RT makes no attempt to correlate process and message priorities and, indeed, has nothing to do with process priorities whatsoever. Such functionality should instead be provided by domain-specific middleware. Such middleware could, for example, set the priority of a process to the priority of the messages it receives. In the absence of middleware, the application developer must manage process priorities explicitly. For example, if the programmer wishes for processes receiving high-priority messages to have a high process priority, the process priority must be set explicitly. Similarly, in the case of request handlers associated with a prioritized message, the user of MPI/RT must explicitly correlate the priority of the handler to that of the message by creating a thread, for example, that has a priority corresponding to that of the message and by allowing that thread to handle the message. Alternatively, the request handler mechanism could be used to alter the priority of the receiving process, providing an implementation of the middleware mechanism described above. Notice that this implies that request handlers must almost certainly be run at a higher priority than any MPI/RT process or thread. Thus, while an MPI/RT *implementation* may need to concern itself with process priorities, the *interface* itself does not. ⊤ (final)

10.5 Best Effort Quality of Service Programming Model

Best effort quality of service does not specify any *constraints* or *requirements* that a user requests from the implementation. This is a request from a user for an implementation to deliver a message or an event, or a handler invocation as fast as possible without jeopardizing the quality of service for other objects[6]. There is no specification for this quality of service, just a constant MPIRT_QOS_BEST_EFFORT.

> *Rationale.* From the real-time requirement point of view the best effort quality of service is equivalent to the deadline of *infinity* or the priority which is smaller than all others. From the real-time constraint point of view there is no timeout specified as part of this quality of service. (*End of rationale.*)

> *Advice to users.* Do not confuse MPIRT_QOS_BEST_EFFORT with MPIRT_TIME_BEST_EFFORT. Users can not use the best effort quality of service with timeout. Users can use event-driven quality of service specifying no requirement and the timeout. For example, for channel specification with timeout only, users can specify MPIRT_QOS_CHANNEL_PRTIEV_CREATE with priority as MPIRT_PRIORITY_IGNORE, with start_event as MPIRT_STRING_NAME_NULL, and with start_time_spec as MPIRT_TIME_BEST_EFFORT. (*End of advice to users.*)

⊥ (final)

[6] For the message delivery, MPI-1 can be viewed as the best effort quality of service per individual message.

Chapter 11

Quality-of-Service Specification

Current Status: First formal vote on Section 11.2 without subsection 11.2.3 passed 7/1/1 in May 98.
FFIrst formal vote on Subsection 11.2.3 passed 8/0/0 in June 98.
First formal vote on Section 11.3 passed 8/0/3 in June 98.
First formal vote on Section 11.4 passed 9/0/0 in June 98.
Second formal vote passed 7/0/3 in July 98.
Third formal vote passed 6/0/3 on 12/3/98.

11.1 Introduction

This section defines four separate objects for quality-of-service (QoS) specification: one for channels, one for handlers, and two for event delivery abstraction objects with separate specification for triggers and for receptors.

The QoS objects for channels, handlers, triggers and receptors have different parameters but all belong to the abstract class of MPIRT_QOS and hence can have values of the predefined constants such as MPIRT_QOS_BEST_EFFORT for the best QoS an implementation can provide without violating other objects' QoS guarantees.

11.2 Channel Quality-of-Service Specification

As described in the previous chapters, QoS for communication contains two specifications, one for QoS_constraints (guards) and one for QoS_requirements. Three QoS objects are defined below:

- one for the time-driven real-time communication paradigm which uses a timing interval to define both QoS_constraint and QoS_requirement specifications,
- one for a combined event-driven-with-priority, real-time communication paradigm where a priority defines the QoS_requirement, and events delivered via the event delivery abstraction (receptors) define the QoS_constraints, and
- one for a combined event-and-time-driven-with-priority, real-time communication paradigm where a priority or the deadline defines the QoS_requirements and events delivered via event delivery abstraction (receptors) for the starting guard of the QoS_constraint and either a timeout or the deadline for the stopping guard of the QoS_constraint.

The aggregate size of the message transmission is needed as part of the QoS in order for MPI/RT to provide QoS guarantees. This information is provided through the buffer's dataspec and count parameters that are passed to a channel description through the channel's in-buffer and out-buffer iterators buffers.

In addition to the QoS_constraints and QoS_requirements, the channel QoS specification can have values of predefined constants such as MPIRT_QOS_BEST_EFFORT, defined above, and also MPIRT_QOS_ACCEPT_ANY, which allows one end of a channel to accept the QoS specification of another end of the channel.

If the QoS for all channel endpoints is of type MPIRT_QOS_CHANNEL_PREV or MPIRT_QOS_CHANNEL_PRTIEV then a single occurrence of the event kind that is specified in the QoS that starts the message transfer, transfers only one message over the channel.

11.2.1 Specification for Time-Driven Channels

The time-driven paradigm specifies the time interval for a message transmission. The time interval is defined by the phase and the deadline. The phase can be specified in absolute time or in relative time with respect to the start of the period using MPI/RT type MPIRT_TIME_SPEC. The deadline can be specified in absolute time or by relative time with respect to the phase. The value of MPIRT_TIME_SPEC_IGNORE for the phase or the deadline indicates that phase or deadline is requested, respectively.

The QOS parameters for the time-driven communication paradigm include the following:

- the period, for periodic message transmission,
- the phase (starting/release time), including the relationship between the phase and the period if one exists (a message transmission does not need to start at the beginning of each period). The MPIRT_TIME_SPEC_IGNORE is not allowed for the phase. The phase must be less than or equal to the period.
- the deadline for a message transmission, including the relationship between the deadline and the phase if one exists (the deadline does not need to coincide with the end of a period).

MPIRT_QOS_CHANNEL_TIME_CREATE (name, period, phase, deadline, qostime)

IN	name	name of the object (MPIRT_STRING_NAME)
IN	period	either MPIRT_PERIOD_IGNORE for a period to be ignored or the time for the duration of the period (MPIRT_TIME)
IN	phase	starting time for a message transmission (MPIRT_TIME_SPEC)
IN	deadline	the deadline for the message transmission (MPIRT_TIME_SPEC)
OUT	qostime	time-driven channel QoS (ref. to MPIRT_QOS_CHANNEL_TIME)

```
int MPIRT_Qos_channel_time_create (MPIRT_String_name name, MPIRT_Time period,
            MPIRT_Time_spec phase, MPIRT_Time_spec deadline,
            MPIRT_Qos_channel_time *qostime)
```

```
MPIRT::Qos_Channel_Time (const MPIRT::String_Name name, MPIRT::Time period,
            MPIRT::Time_Spec phase, MPIRT::Time_Spec deadline)
```

If period is specified, not MPIRT_TIME_IGNORE or MPIRT_TIME_NULL, then the phase must be of type MPIRT_RELATIVE. This specifies that the phase is relative to the start of the period.

If phase is of type MPIRT_ABSOLUTE then period must be MPIRT_TIME_IGNORE. If deadline is specified as absolute then phase must be also of type MPIRT_ABSOLUTE and period must be MPIRT_TIME_IGNORE. For all other cases deadline must be of type MPIRT_RELATIVE which specifies the deadline relative to phase. Notice that deadline can be smaller, equal to, or even larger than the period.

The next three operations query the parameters of the time-driven channel QoS.

MPIRT_QOS_CHANNEL_TIME_GET_PERIOD (qostime, period)

IN	qostime	time-driven channel quality-of-service (MPIRT_QOS_CHANNEL_TIME)
OUT	period	either MPIRT_PERIOD_IGNORE for a period to be ignored or the time for the duration of the period (ref. to MPIRT_TIME)

```
int MPIRT_Qos_channel_time_get_period (MPIRT_Qos_channel_time qostime,
              MPIRT_Time *period)
```

```
int MPIRT::Qos_Channel_Time::Get_Period (MPIRT::Time &period) const
```

MPIRT_QOS_CHANNEL_TIME_GET_PHASE (qostime, phase)

IN	qostime	time-driven channel QoS (MPIRT_QOS_CHANNEL_TIME)
OUT	phase	starting time for a message transmission (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_time_get_phase (MPIRT_Qos_channel_time qostime,
              MPIRT_Time_spec *phase)
```

```
int MPIRT::Qos_Channel_Time::Get_Phase (MPIRT::Time_Spec &phase) const
```

MPIRT_QOS_CHANNEL_TIME_GET_DEADLINE (qostime, deadline)

IN	qostime	time-driven channel QoS (MPIRT_QOS_CHANNEL_TIME)
OUT	deadline	the deadline for a message transmission (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_time_get_deadline (MPIRT_Qos_channel_time qostime,
              MPIRT_Time_spec *deadline)
```

```
int MPIRT::Qos_Channel_Time::Get_Deadline (MPIRT::Time_Spec &deadline) const
```

The next three operations set the parameters of the time-driven channel QoS.

MPIRT_QOS_CHANNEL_TIME_SET_PERIOD (qostime, period)

INOUT	qostime	time-driven channel QoS (ref. to MPIRT_QOS_CHANNEL_TIME)
IN	period	either MPIRT_TIME_IGNORE for a period to be ignored or a time for the duration of the period (MPIRT_TIME)

```
int MPIRT_Qos_channel_time_set_period (MPIRT_Qos_channel_time *qostime,
              MPIRT_Time period)
```

```
int MPIRT::Qos_Channel_Time::Set_Period (const MPIRT::Time period)
```

MPIRT_QOS_CHANNEL_TIME_SET_PHASE (qostime, phase)

INOUT	qostime	time-driven channel QoS (ref. to MPIRT_QOS_CHANNEL_TIME)
IN	phase	starting time for a message transmission (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_time_set_phase (MPIRT_Qos_channel_time *qostime,
              MPIRT_Time_spec phase)
```

```
int MPIRT::Qos_Channel_Time::Set_Phase (const MPIRT::Time_Spec phase)
```

MPIRT_QOS_CHANNEL_TIME_SET_DEADLINE (qostime, deadline)

INOUT	qostime	time-driven channel QoS (ref. to MPIRT_QOS_CHANNEL_TIME)
IN	deadline	the deadline for the message transmission (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_time_set_deadline (MPIRT_Qos_channel_time *qostime,
              MPIRT_time_spec deadline)
```

```
int MPIRT::Qos_Channel_Time::Set_Deadline (const MPIRT::Time_Spec deadline)
```

The duplication, freeing, equality testing, and the query and set operations for the name parameter of the time-driven channel QoS are derived from the base class MPIRT_OBJECT (see Section 2.2).

⊥ (final2)

11.2.2 Specification for Event-Driven-with-Priority Channels

The event-driven-with-priority real-time communication paradigm uses priority for the QoS requirement specification. The QoS_constraints are specified as events delivered by implicit receptors. The receptors are defined by event names, event scopes, event priorities, and event time specifications, and delivered events guard (start and stop) a message transfer over the channel. If the event_name is MPIRT_STRING_NAME_NULL, then no starting or timeout event is specified.

The QoS parameters for the event-driven-with-priority paradigm include the following:

- integer priority of a channel,
- event name for the implicit receptor for the delivery of events that start a message transfer over the channel,
- scope for the implicit receptor for the delivery of events that start a message transfer over the channel,
- priority for the implicit receptor for the delivery of events that start a message transfer over the channel,
- time specification for the deadline or the timeout for the implicit receptor for the delivery of events that start a message transfer over the channel,
- event name for the implicit receptor for the delivery of events that stop a message transfer over the channel if it is not finished yet,
- scope for the implicit receptor for the delivery of events that stop a message transfer over the channel if it is not finished yet,
- priority for the implicit receptor for the delivery of events that stop a message transfer over the channel if it is not finished yet, and
- time specification for the deadline or the timeout for the implicit receptor for the delivery of events that stop a message transfer over the channel if it is not finished yet.

The event-driven-with-priority channel QoS object is defined below. $\top$ (final2)

MPIRT_QOS_CHANNEL_PREV_CREATE (name, priority, start_event, start_scope, start_priority, start_time_spec, stop_event, stop_scope, stop_priority, stop_time_spec, qosprev)

IN	name	name of the object (MPIRT_STRING_NAME)
IN	priority	integer priority of a channel or a constant MPIRT_PRIORITY_IGNORE (integer)
IN	start_event	either an event name for the implicit receptor for events starting a channel message transfer or MPIRT_STRING_NAME_NULL for no starting event (MPIRT_STRING_NAME)
IN	start_scope	group that defines the scope for the implicit receptor for delivery of the events that start a channel message transfer (MPIRT_GROUP)
IN	start_priority	either a priority or the constant MPIRT_PRIORITY_IGNORE for no priority for the implicit receptor for the delivery of the events that start a channel message transfer (integer)
IN	start_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of the events that start a channel message transfer (MPIRT_TIME_SPEC)
IN	stop_event	either an event name for the implicit receptor for events that stop a channel message transfer if it has not finished yet or MPIRT_STRING_NAME_NULL for no stopping event (MPIRT_STRING_NAME)
IN	stop_scope	a group that defines the scope for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (MPIRT_GROUP)
IN	stop_priority	either a priority or the constant MPIRT_PRIORITY_IGNORE for no priority for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (integer)
IN	stop_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (MPIRT_TIME_SPEC)
OUT	qosprev	QoS for an event-driven with priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)

```
int MPIRT_Qos_channel_prev_create (MPIRT_String_name name, int priority,
            MPIRT_String_name start_event, MPIRT_Group start_scope,
            int start_priority, MPIRT_Time_spec start_time_spec,
            MPIRT_String_name stop_event, MPIRT_Group stop_scope,
            int stop_priority, MPIRT_Time_spec stop_time_spec,
            MPIRT_Qos_channel_prev *qosprev)
```

```
MPIRT::Qos_Channel_Prev (const MPIRT::String_Name name, int priority, const
            MPIRT::String_Name start_event, const MPIRT::Group &start_scope,
            int start_priority, MPIRT::Time_Spec start_time_spec, const
            MPIRT::String_Name stop_event, const MPIRT::Group &stop_scope,
            int stop_priority, MPIRT::Time_Spec stop_time_spec)
```

The next operations query individual parameters of the event-driven-with-priority channel QoS.

MPIRT_QOS_CHANNEL_PREV_GET_PRIORITY (qosprev, priority)

IN	qosprev	QoS object for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	priority	integer priority of a channel or the constant MPIRT_PRIORITY_IGNORE for no priority constraint (ref. to integer)

```
int MPIRT_Qos_channel_prev_get_priority (MPIRT_Qos_channel_prev qosprev,
            int *priority)
```

```
int MPIRT::Qos_Channel_Prev::Get_Priority (int &priority) const
```

MPIRT_QOS_CHANNEL_PREV_GET_START_EVENT (qosprev, start_event)

IN	qosprev	QoS object for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	start_event	either an event name for the implicit receptor for events starting a channel message transfer or MPIRT_STRING_NAME_NULL for no starting event (ref. to MPIRT_STRING_NAME)

```
int MPIRT_Qos_channel_prev_get_start_event (MPIRT_Qos_channel_prev qosprev,
            MPIRT_String_name *start_event)
```

```
int MPIRT::Qos_Channel_Prev::Get_Start_Event (MPIRT::String_Name &start_event)
            const
```

MPIRT_QOS_CHANNEL_PREV_GET_START_SCOPE (qosprev, start_scope)

IN	qosprev	QoS object for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	start_scope	a group that defines the scope for the implicit receptor for the delivery of the events that start a channel message transfer (ref. to MPIRT_GROUP)

```
int MPIRT_Qos_channel_prev_get_start_scope (MPIRT_Qos_channel_prev qosprev,
            MPIRT_Group *start_scope)
```

```
int MPIRT::Qos_Channel_Prev::Get_Start_Scope (MPIRT::Group *&start_scope) const
```

MPIRT_QOS_CHANNEL_PREV_GET_START_PRIORITY (qosprev, start_priority)

IN	qosprev	QoS object for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	start_priority	either a priority or the constant MPIRT_PRIORITY_IGNORE for no priority for the implicit receptor for the delivery of the events that start a channel message transfer (ref. to integer)

```
int MPIRT_Qos_channel_prev_get_start_priority (MPIRT_Qos_channel_prev qosprev,
            int *start_priority)
```

```
int MPIRT::Qos_Channel_Prev::Get_Start_Priority (int &start_priority) const
```

MPIRT_QOS_CHANNEL_PREV_GET_START_TIME_SPEC (qosprev, start_time_spec)

IN	qosprev	QoS object for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	start_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of events that start a channel message transfer (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prev_get_start_time_spec (MPIRT_Qos_channel_prev qosprev,
            MPIRT_Time_spec *start_time_spec)
```

```
int MPIRT::Qos_Channel_Prev::Get_Start_Time_Spec
            (MPIRT::Time_Spec &start_time_spec) const
```

MPIRT_QOS_CHANNEL_PREV_GET_STOP_EVENT (qosprev, stop_event)

IN	qosprev	QoS for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	stop_event	either an event name for the implicit receptor for events that stop a channel message transfer if it has not finished yet or MPIRT_STRING_NAME_NULL for no stopping event (ref. to MPIRT_STRING_NAME)

```
int MPIRT_Qos_channel_prev_get_stop_event (MPIRT_Qos_channel_prev qosprev,
            MPIRT_String_name *stop_event)
```

```
int MPIRT::Qos_Channel_Prev::Get_Stop_Event (MPIRT::String_Name &stop_event) const
```

MPIRT_QOS_CHANNEL_PREV_GET_STOP_SCOPE (qosprev, stop_scope)

IN	qosprev	QoS object for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	stop_scope	a group that defines the scope for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (ref. to MPIRT_GROUP)

```
int MPIRT_Qos_channel_prev_get_stop_scope (MPIRT_Qos_channel_prev qosprev,
              MPIRT_Group *stop_scope)
```

```
int MPIRT::Qos_Channel_Prev::Get_Stop_Scope (MPIRT::Group *&stop_scope) const
```

MPIRT_QOS_CHANNEL_PREV_GET_STOP_PRIORITY (qosprev, stop_priority)

IN	qosprev	QoS object for an event-driven with priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	stop_priority	either a priority or the constant MPIRT_PRIORITY_IGNORE for no priority for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (ref. to integer)

```
int MPIRT_Qos_channel_prev_get_stop_priority (MPIRT_Qos_channel_prev qosprev,
              int *stop_priority)
```

```
int MPIRT::Qos_Channel_Prev::Get_Stop_Priority (int &stop_priority) const
```

MPIRT_QOS_CHANNEL_PREV_GET_STOP_TIME_SPEC (qosprev, stop_time_spec)

IN	qosprev	QoS object for an event-driven-with-priority channel (MPIRT_QOS_CHANNEL_PREV)
OUT	stop_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prev_get_stop_time_spec (MPIRT_Qos_channel_prev qosprev,
              MPIRT_Time_spec *stop_time_spec)
```

```
int MPIRT::Qos_Channel_Prev::Get_Stop_Time_Spec (MPIRT::Time_Spec &stop_time_spec)
              const
```

The next operations set individual parameters of the event-driven-with-priority channel QoS.

MPIRT_QOS_CHANNEL_PREV_SET_PRIORITY (qosprev, priority)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	priority	integer priority of a channel or a constant MPIRT_PRIORITY_IGNORE for no priority constraint (integer)

```
int MPIRT_Qos_channel_prev_set_priority (MPIRT_Qos_channel_prev *qosprev,
              int priority)
```

```
int MPIRT::Qos_Channel_Prev::Set_Priority (int priority)
```

MPIRT_QOS_CHANNEL_PREV_SET_START_EVENT (qosprev, start_event)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	start_event	either an event name for the implicit receptor for events starting a channel message transfer or MPIRT_STRING_NAME_NULL for no starting event (MPIRT_STRING_NAME)

```
int MPIRT_Qos_channel_prev_set_start_event (MPIRT_Qos_channel_prev *qosprev,
              MPIRT_String_name start_event)
```

```
int MPIRT::Qos_Channel_Prev::Set_Start_Event (const MPIRT::String_Name start_event)
```

MPIRT_QOS_CHANNEL_PREV_SET_START_SCOPE (qosprev, start_scope)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	start_scope	group that defines the scope for the implicit receptor for the delivery of the events that start a channel message transfer (MPIRT_GROUP)

```
int MPIRT_Qos_channel_prev_set_start_scope (MPIRT_Qos_channel_prev *qosprev,
              MPIRT_Group start_scope)
```

```
int MPIRT::Qos_Channel_Prev::Set_Start_Scope (const MPIRT::Group &start_scope)
```

MPIRT_QOS_CHANNEL_PREV_SET_START_PRIORITY (qosprev, start_priority)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	start_priority	either a priority or the constant MPIRT_PRIORITY_IGNORE for no priority for the implicit receptor for the delivery of the events that start a channel message transfer (integer)

```
int MPIRT_Qos_channel_prev_set_start_priority (MPIRT_Qos_channel_prev *qosprev,
              int start_priority)
```

```
int MPIRT::Qos_Channel_Prev::Set_Start_Priority (int start_priority)
```

MPIRT_QOS_CHANNEL_PREV_SET_START_TIME_SPEC (qosprev, start_time_spec)

INOUT	qosprev	QoS object for an event-driven with priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	start_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of the events that start a channel message transfer (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prev_set_start_time_spec (MPIRT_Qos_channel_prev *qosprev,
              MPIRT_Time_spec start_time_spec)
```

```
int MPIRT::Qos_Channel_Prev::Set_Start_Time_Spec (const
              MPIRT::Time_Spec &start_time_spec)
```

MPIRT_QOS_CHANNEL_PREV_SET_STOP_EVENT (qosprev, stop_event)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	stop_event	either an event name for implicit receptor for events that stop a channel message transfer if it has not finished yet or MPIRT_STRING_NAME_NULL for no stopping event (MPIRT_STRING_NAME)

```
int MPIRT_Qos_channel_prev_set_stop_event (MPIRT_Qos_channel_prev *qosprev,
              MPIRT_String_name stop_event)
```

```
int MPIRT::Qos_Channel_Prev::Set_Stop_Event (const MPIRT::String_Name stop_event)
```

MPIRT_QOS_CHANNEL_PREV_SET_STOP_SCOPE (qosprev, stop_scope)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	stop_scope	a group that defines the scope for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (MPIRT_GROUP)

```
int MPIRT_Qos_channel_prev_set_stop_scope (MPIRT_Qos_channel_prev *qosprev,
             MPIRT_Group stop_scope)
```

```
int MPIRT::Qos_Channel_Prev::Set_Stop_Scope (const MPIRT::Group &stop_scope)
```

MPIRT_QOS_CHANNEL_PREV_SET_STOP_PRIORITY (qosprev, stop_priority)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	stop_priority	either a priority or the constant MPIRT_PRIORITY_IGNORE for no priority for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (integer)

```
int MPIRT_Qos_channel_prev_set_stop_priority (MPIRT_Qos_channel_prev *qosprev,
             int stop_priority)
```

```
int MPIRT::Qos_Channel_Prev::Set_Stop_Priority (int stop_priority)
```

MPIRT_QOS_CHANNEL_PREV_SET_STOP_TIME_SPEC (qosprev, stop_time_spec)

INOUT	qosprev	QoS object for an event-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PREV)
IN	stop_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of the events that stop a channel message transfer if it has not finished yet (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prev_set_stop_time_spec (MPIRT_Qos_channel_prev *qosprev,
             MPIRT_Time_spec stop_time_spec)
```

```
int MPIRT::Qos_Channel_Prev::Set_Stop_Time_Spec (const
             MPIRT::Time_Spec &stop_time_spec)
```

⊥ (final2)
⊤ (final)
⊥ (final)

The duplication, freeing, equality testing, and query and set operations for parameter name for event-driven-with-priority channel QoS objects are derived from the base class MPIRT_OBJECT (see Section 2.2).

11.2.3 Specification for Combined Event and Time-Driven with Priority Channels

The event-and-time-driven-with-priority, real-time communication paradigm specification has two ways to define a QoS_requirement. One is to use a priority for the QoS_requirement specification, and the other is to use the deadline as both a QoS_requirement and as a timeout guard as QoS_constraint. The stop guard for the QoS_constraint is specified in MPI/RT type MPIRT_TIME_SPEC. If the QoS_requirement is specified as a priority, then the stop guard for QoS_constraint represents a timeout and not a deadline. If the priority value is the constant MPIRT_PRIORITY_IGNORE, then the stop guard specifies the deadline that represents both the QoS_requirement and the stop guard of the QoS_constraint. The start guard is specified as events delivered by an implicit receptor. The receptor is defined by the event name, event scope, event priority, and the event time specification, and delivered events start a message transfer over the channel. If the event_name is MPIRT_STRING_NAME_NULL, then no starting event is specified.

The QOS parameters for the event-and-time-driven-with-priority paradigm include the following:

- integer priority of a channel,
- event name for the implicit receptor for the delivery of events that start a message transfer over the channel,
- scope for the implicit receptor for the delivery of events that start a message transfer over the channel,
- priority for the implicit receptor for the delivery of events that start a message transfer over the channel,
- time specification for the deadline or the timeout for the implicit receptor for the delivery of events that start a message transfer over the channel, and
- time specification for a timeout if the priority is specified, or the deadline if the priority has a value of MPIRT_PRIORITY_IGNORE for the channel message transfer.

The object for QoS for an event-and-time-driven-with-priority channel is defined below. ⊤ (final2)

MPIRT_QOS_CHANNEL_PRTIEV_CREATE (name, priority, start_event, start_scope, start_priority, start_time_spec, stop_spec, qos)

IN	name	name of the object (MPIRT_STRING_NAME)
IN	priority	integer priority of a channel or a constant MPIRT_PRIORITY_IGNORE (integer)
IN	start_event	either an event name for starting a channel message transmission or MPIRT_STRING_NAME_NULL for no starting event (MPIRT_STRING_NAME)
IN	start_scope	group that defines the scope for the implicit start event receptor (MPIRT_GROUP)
IN	start_priority	either a priority for the start event delivery to the implicit start event receptor or the constant MPIRT_PRIORITY_IGNORE for no priority for the start event delivery (integer)
IN	start_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of the events that start a channel message transfer (MPIRT_TIME_SPEC)
IN	stop_spec	time specification for a timeout event if priority is requested or a deadline if priority is MPIRT_PRIORITY_IGNORE for a channel message transfer (MPIRT_TIME_SPEC)
OUT	qos	QoS for an event-and-time-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PRTIEV)

```
int MPIRT_Qos_channel_prtiev_create (MPIRT_String_name name, int priority,
            MPIRT_String_name start_event, MPIRT_Group start_scope,
            int start_priority, MPIRT_Time_spec start_time_spec,
            MPIRT_Time_spec stop_spec, MPIRT_Qos_channel_prtiev *qos)
```

```
MPIRT::Qos_Channel_Prtiev (const MPIRT::String_Name name, int priority, const
            MPIRT::String_Name start_event, const MPIRT::Group &start_scope,
            int start_priority, MPIRT::Time_Spec start_time_spec,
            MPIRT::Time_Spec stop_spec)
```

The next operations query individual parameters of the event-and-time-driven-with-priority channel QoS.

MPIRT_QOS_CHANNEL_PRTIEV_GET_PRIORITY (qos, priority)

IN	qos	quality-of-service (QoS) object for an event-and-time-driven-with-priority channel (MPIRT_QOS_CHANNEL_PRTIEV)
OUT	priority	integer priority of a channel or the constant MPIRT_PRIORITY_IGNORE for no priority constraint (ref. to integer)

```
int MPIRT_Qos_channel_prtiev_get_priority (MPIRT_Qos_channel_prtiev qos,
            int *priority)
```

```
int MPIRT::Qos_Channel_Prtiev::Get_Priority (int &priority) const
```

MPIRT_QOS_CHANNEL_PRTIEV_GET_START_EVENT (qos, start_event)

IN	qos	QoS object for an event-and-time-driven-with-priority channel (MPIRT_QOS_CHANNEL_PRTIEV)
OUT	start_event	either event name for starting a channel message transmission or MPIRT_STRING_NAME_NULL for no starting event (ref. to MPIRT_STRING_NAME)

```
int MPIRT_Qos_channel_prtiev_get_start_event (MPIRT_Qos_channel_prtiev qos,
              MPIRT_String_name *start_event)
```

```
int MPIRT::Qos_Channel_Prtiev::Get_Start_Event (MPIRT::String_Name &start_event)
              const
```

MPIRT_QOS_CHANNEL_PRTIEV_GET_START_SCOPE (qos, start_scope)

IN	qos	QoS object for an event-and-time-driven-with-priority channel (MPIRT_QOS_CHANNEL_PRTIEV)
OUT	start_scope	group that defines the scope for the implicit start event receptor (ref. to MPIRT_GROUP)

```
int MPIRT_Qos_channel_prtiev_get_start_scope (MPIRT_Qos_channel_prtiev qos,
              MPIRT_Group *start_scope)
```

```
int MPIRT::Qos_Channel_Prtiev::Get_Start_Scope (MPIRT::Group *&start_scope) const
```

MPIRT_QOS_CHANNEL_PRTIEV_GET_START_PRIORITY (qos, start_priority)

IN	qos	QoS object for an event-and-time-driven-with-priority channel (MPIRT_QOS_CHANNEL_PRTIEV)
OUT	start_priority	either a priority for the start event delivery to the implicit start event receptor or the constant MPIRT_PRIORITY_IGNORE for no priority for the start event delivery (ref. to integer)

```
int MPIRT_Qos_channel_prtiev_get_start_priority (MPIRT_Qos_channel_prtiev qos,
              int *start_priority)
```

```
int MPIRT::Qos_Channel_Prtiev::Get_Start_Priority (int &start_priority) const
```

MPIRT_QOS_CHANNEL_PRTIEV_GET_START_TIME_SPEC (qos, start_time_spec)

IN	qos	QoS object for an event-and-time-driven-with-priority channel (MPIRT_QOS_CHANNEL_PRTIEV)
OUT	start_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of events that start a channel message transfer (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prtiev_get_start_time_spec (MPIRT_Qos_channel_prtiev qos,
            MPIRT_Time_spec *start_time_spec)
```

```
int MPIRT::Qos_Channel_Prtiev::Get_Start_Time_Spec
            (MPIRT::Time_Spec &start_time_spec) const
```

MPIRT_QOS_CHANNEL_PRTIEV_GET_STOP_SPEC (qos, stop_spec)

IN	qos	QoS for an event-and-time-driven-with-priority channel (MPIRT_QOS_CHANNEL_PRTIEV)
OUT	stop_spec	time specification for a channel timeout event if priority is requested or a deadline if priority is MPIRT_PRIORITY_IGNORE (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prtiev_get_stop_spec (MPIRT_Qos_channel_prtiev qos,
            MPIRT_time_spec *stop_spec)
```

```
int MPIRT::Qos_Channel_Prtiev::Get_Stop_Spec (MPIRT::Time_Spec &stop_spec) const
```

The next operations set individual parameters of the event-and-time-driven-with-priority channel QoS.

MPIRT_QOS_CHANNEL_PRTIEV_SET_PRIORITY (qos, priority)

INOUT	qos	event-and-time-driven-with-priority channel QoS (ref. to MPIRT_QOS_CHANNEL_PRTIEV)
IN	priority	integer priority of a channel or a constant MPIRT_PRIORITY_IGNORE for no priority constraint (integer)

```
int MPIRT_Qos_channel_prtiev_set_priority (MPIRT_Qos_channel_prtiev *qos,
            int priority)
```

```
int MPIRT::Qos_Channel_Prtiev::Set_Priority (int priority)
```

MPIRT_QOS_CHANNEL_PRTIEV_SET_START_EVENT (qos, start_event)

INOUT	qos	event-and-time-driven-with-priority channel QoS (ref. to MPIRT_QOS_CHANNEL_PRTIEV)
IN	start_event	either an event name for the implicit receptor for events that start a channel message transfer or MPIRT_STRING_NAME_NULL for no starting event (MPIRT_STRING_NAME)

```
int MPIRT_Qos_channel_prtiev_set_start_event (MPIRT_Qos_channel_prtiev *qos,
              MPIRT_String_name start_event)
```

```
int MPIRT::Qos_Channel_Prtiev::Set_Start_Event (const
              MPIRT::String_Name start_event)
```

MPIRT_QOS_CHANNEL_PRTIEV_SET_START_SCOPE (qos, start_scope)

INOUT	qos	QoS object for an event-and-time-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PRTIEV)
IN	start_scope	group that defines the scope for the implicit receptor for delivery of the events that start a channel message transfer (ref. to MPIRT_GROUP)

```
int MPIRT_Qos_channel_prtiev_set_start_scope (MPIRT_Qos_channel_prtiev *qos,
              MPIRT_Group start_scope)
```

```
int MPIRT::Qos_Channel_Prtiev::Set_Start_Scope (const MPIRT::Group &start_scope)
```

MPIRT_QOS_CHANNEL_PRTIEV_SET_START_PRIORITY (qos, start_priority)

INOUT	qos	QoS object for an event-and-time-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PRTIEV)
IN	start_priority	either a priority or the constant MPIRT_PRIORITY_IGNORE for no priority for the implicit receptor for the delivery of the events that start a channel message transfer (integer)

```
int MPIRT_Qos_channel_prtiev_set_start_priority (MPIRT_Qos_channel_prtiev *qos,
              int start_priority)
```

```
int MPIRT::Qos_Channel_Prtiev::Set_Start_Priority (int start_priority)
```

MPIRT_QOS_CHANNEL_PRTIEV_SET_START_TIME_SPEC (qos, start_time_spec)

INOUT	qos	QoS object for an event-and-time-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PRTIEV)
IN	start_time_spec	either a deadline, if start_priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the implicit receptor for the delivery of events that start a channel message transfer (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prtiev_set_start_time_spec (MPIRT_Qos_channel_prtiev *qos,
              MPIRT_Time_spec start_time_spec)
```

```
int MPIRT::Qos_Channel_Prtiev::Set_Start_Time_Spec
              (MPIRT::Time_Spec start_time_spec)
```

MPIRT_QOS_CHANNEL_PRTIEV_SET_STOP_SPEC (qos, stop_spec)

INOUT	qos	QoS object for an event-and-time-driven-with-priority channel (ref. to MPIRT_QOS_CHANNEL_PRTIEV)
IN	stop_spec	time specification for a channel timeout event if priority is requested or a channel message transfer deadline if priority is MPIRT_PRIORITY_IGNORE (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_channel_prtiev_set_stop_spec (MPIRT_Qos_channel_prtiev *qosprev,
              MPIRT_Time_spec stop_spec)
```

```
int MPIRT::Qos_Channel_Prtiev::Set_Stop_Spec (MPIRT::Time_Spec stop_spec)
```

Duplication, freeing, comparison, and query and set operations for name for QoS for the event-and-time-driven-with-priority channel are derived from the base class MPIRT_OBJECT (see Section 2.2).

11.3 Handler Quality-of-Service Specification

The QoS for a handler object contains only a priority as a QoS_requirement and a timeout as a QoS_constraint. The handler QoS does not define a start for the handler execution since a handler is attached to an event receptor and is invoked automatically (by implementation) upon an event arrival. The QoS requirement for a handler can only be of the priority form. The timeout is of the type MPIRT_TIME_SPEC of the MPIRT_RELATIVE form. For the synchronous handlers the priority must be MPIRT_PRIORITY_INHERIT to represent that the synchronous handlers run in the context of the receptor to which the handlers are "attached" to. An asynchronous handlers can also define priority as MPIRT_PRIORITY_INHERIT which means that they run in their own thread(s) with the same priority as the receptor thread.

Rationale. While a channel can have a deadline for a QoS_requirement, a handler cannot. The reason for this is that an implementation does not know the execution time for the handler and can not determine whether the requested deadline can be met at admission test time. The handler is part of the user code, and only the user knows its execution duration. In contrast, the channel message transfer is performed by the implementation. Hence, the implementation can determine the execution time based upon message size and the communication traffic known at the admission test time (commit). (*End of rationale.*)

MPIRT_QOS_HANDLER_CREATE (name, priority, timeout, qos_handler)

IN	name	name of the object (MPIRT_STRING_NAME)
IN	priority	priority of a handler execution or a constant MPIRT_PRIORITY_INHERIT (integer)
IN	timeout	the timeout for the handler execution (MPIRT_TIME_SPEC)
OUT	qos_handler	handler QoS (ref. to MPIRT_QOS_HANDLER)

```
int MPIRT_Qos_handler_create (MPIRT_String_name name, int priority,
              MPIRT_Time_spec timeout, MPIRT_Qos_handler *qos_handler)
```

```
MPIRT::Qos_Handler (const MPIRT::String_Name name, int priority,
              MPIRT::Time_Spec timeout)
```

The next two operations query the parameters of the handler QoS.

MPIRT_QOS_HANDLER_GET_PRIORITY (qos_handler, priority)

IN	qos_handler	handler QoS (MPIRT_QOS_HANDLER)
OUT	priority	priority of the handler execution (ref. to integer)

```
int MPIRT_Qos_handler_get_priority (MPIRT_Qos_handler qos_handler, int *priority)
```

```
int MPIRT::Qos_Handler::Get_Priority (int &priority) const
```

MPIRT_QOS_HANDLER_GET_TIMEOUT (qos_handler, timeout)

IN	qos_handler	handler QoS (MPIRT_QOS_HANDLER)
OUT	timeout	the timeout for the handler execution (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_handler_get_timeout (MPIRT_Qos_handler qos_handler,
              MPIRT_Time_spec *timeout)
```

```
int MPIRT::Qos_Handler::Get_Timeout (MPIRT::Time_Spec &timeout) const
```

The next two operations set the parameters of the handler QoS object.

MPIRT_QOS_HANDLER_SET_PRIORITY (qos_handler, priority)

INOUT	qos_handler	handler QoS (ref. to MPIRT_QOS_HANDLER)
IN	priority	priority of the handler execution (integer)

```
int MPIRT_Qos_handler_set_priority (MPIRT_Qos_handler *qos_handler, int priority)
```

```
int MPIRT::Qos_Handler::Set_Priority (int priority)
```

MPIRT_QOS_HANDLER_SET_TIMEOUT (qos_handler, timeout)

INOUT	qos_handler	handler QoS (ref. to MPIRT_QOS_HANDLER)
IN	timeout	the timeout for the handler execution (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_handler_set_timeout (MPIRT_Qos_handler *qos_handler,
             MPIRT_Time_spec timeout)
```

```
int MPIRT::Qos_Handler::Set_Timeout (MPIRT::Time_spec timeout)
```

Duplication, freeing, comparison, and query and set operations for name parameter for handler QoS are derived from the base class MPIRT_OBJECT (see Section 2.2).

11.4 Event Delivery Abstraction's Quality-of-Service Specification

There are two objects that comprise the event delivery abstraction: triggers and receptors (see Chapter 4.) Both of these objects have a QoS specification.

11.4.1 Quality-of-Service for Triggers

The QoS for event triggers defines the "rate" for the event generation that, together with an event receptor's QoS, allows MPI/RT to determine if events can be delivered from the triggers to the receptors within the required QoS specifications.

The rate is defined by three parameters:

Bound - the minimal (lower bound) inter-event generation time. The minimal time between any two event occurrences.

Period - a time interval.

Occurrence - the maximum number (upper bound) of the event occurrences (explicit and implicit MPIRT_TRIGGER_RAISEs) within any time interval smaller than or equal to the period.

The bound parameter allows users to define how closely in time two events can be generated. It allows the implementation to determine how much resource usage is needed for delivery of events from this trigger for small time intervals. If the bound is not known, users can specify MPIRT_TIME_IGNORE. This parameter is orthogonal to the other two parameters.

The period and occurrence parameters allow users to specify the maximum number of event occurrences over any time interval. For the case when there is only one occurrence of the event per period, this specification is analogous to the channel period specification. However, if an event does not necessarily occur every period, or if there is nested periodicity for event generation, the standard periodic specification that is used by channel QoS is overly pessimistic and will require more resources than needed. If the period is not known, users can specify MPIRT_TIME_IGNORE. In ⊤ (final2) this case occurrence is ignored. ⊥ (final2)

For example, if the event is raised exactly every two milliseconds without any skew, the specification is $bound = period = 2$ milliseconds and $occurrence = 1$. If an event can occur with a periodicity of 2 milliseconds, but the exact time of the event can fluctuate within the period such that the time difference between two occurrences of the event that are k periods apart is at most $k \times period + 1$ milliseconds and at least $k \times period - 1$, then the specification is as follows: $bound = 1$ millisecond, $period = 2$ millisecond, and $occurrence = 2$.

This specification allows the implementation to partially determine the load to the system for this event name (the other half of the load specification comes from the event receptors).

MPIRT_QOS_TRIGGER_CREATE (name, bound, period, occurrence, qos_trigger)

IN	name	name of the object (MPIRT_STRING_NAME)
IN	bound	the minimal inter-event generation time (MPIRT_TIME)
IN	period	period of the event generation or MPIRT_TIME_IGNORE if no inter-trigger event time is known (MPIRT_TIME) ⊤ (final2) ⊥ (final2)
IN	occurrence	the maximum number of event occurrences per period (integer)
OUT	qos_trigger	event trigger QoS object (ref. to MPIRT_QOS_TRIGGER)

```
int MPIRT_Qos_trigger_create (MPIRT_String_name name, MPIRT_Time bound,
              MPIRT_Time period, int occurrence, MPIRT_Qos_trigger *qos_trigger)
```

```
MPIRT::Qos_Trigger (const MPIRT::String_Name name, MPIRT::Time bound,
              MPIRT::Time period, int occurrence)
```

The next three operations query the parameters of the event trigger QoS object.

MPIRT_QOS_TRIGGER_GET_BOUND (qos_trigger, bound)

IN	qos_trigger	event trigger QoS object (MPIRT_QOS_TRIGGER)
OUT	bound	the minimal inter-event generation time (ref. to MPIRT_TIME)

```
int MPIRT_Qos_trigger_get_bound (MPIRT_Qos_trigger qos_trigger, MPIRT_Time *bound)
```

```
int MPIRT::Qos_Trigger::Get_Bound (MPIRT::Time &bound) const
```

MPIRT_QOS_TRIGGER_GET_PERIOD (qos_trigger, period)

IN	qos_trigger	event trigger QoS object (MPIRT_QOS_TRIGGER)
OUT	period	period of the event generation or MPIRT_TIME_IGNORE if no inter-trigger event time is known (ref. to MPIRT_TIME)

```
int MPIRT_Qos_trigger_get_period (MPIRT_Qos_trigger qos_trigger,
              MPIRT_Time *period)
```

```
int MPIRT::Qos_Trigger::Get_Period (MPIRT::Time &period) const
```

MPIRT_QOS_TRIGGER_GET_OCCURRENCE (qos_trigger, occurrence)

IN	qos_trigger	event trigger QoS object (MPIRT_QOS_TRIGGER)
OUT	occurrence	the maximum number of event occurrences per period (ref. to integer)

```
int MPIRT_Qos_trigger_get_occurrence (MPIRT_Qos_trigger qos_trigger,
              int *occurrence)
```

```
int MPIRT::Qos_Trigger::Get_Occurrence (int &occurrence) const
```

The next three operations set the parameters of the event trigger QoS object.

MPIRT_QOS_TRIGGER_SET_BOUND (qos_trigger, bound)

INOUT	qos_trigger	event trigger QoS object (ref. to MPIRT_QOS_TRIGGER)
IN	bound	the minimal inter-event generation time (MPIRT_TIME)

```
int MPIRT_Qos_trigger_set_bound (MPIRT_Qos_trigger *qos_trigger, MPIRT_Time bound)
```

```
int MPIRT::Qos_Trigger::Set_Bound (const MPIRT::Time &bound)
```

MPIRT_QOS_TRIGGER_SET_PERIOD (qos_trigger, period)

INOUT	qos_trigger	event trigger QoS object (ref. to MPIRT_QOS_TRIGGER)
IN	period	period of the event generation or MPIRT_TIME_IGNORE if no inter-trigger event time is known (MPIRT_TIME)

```
int MPIRT_Qos_trigger_set_period (MPIRT_Qos_trigger *qos_trigger,
              MPIRT_Time period)
```

```
int MPIRT::Qos_Trigger::Set_Period (MPIRT::Time period)
```

MPIRT_QOS_TRIGGER_SET_OCCURRENCE (qos_trigger, occurrence)

INOUT	qos_trigger	event trigger QoS object (ref. to MPIRT_QOS_TRIGGER)
IN	occurrence	the maximum number of event occurrences per period (integer)

```
int MPIRT_Qos_trigger_set_occurrence (MPIRT_Qos_trigger qos_trigger,
              int occurrence)
```

```
int MPIRT::Qos_Trigger::Set_Occurrence (int occurrence)
```

Duplication, freeing, comparison, and query and set operations for name parameter of the QoS of event triggers are derived from the base class MPIRT_OBJECT (see Section 2.2).

11.4.2 Quality-of-Service for Receptors

The QoS for event receptors defines the user requests for the delivery of events from triggers to receptors. This specification delivery is analogous in specification to the handler QoS, but the handler QoS puts requirements on computational resources only, while the event receptor QoS puts requirements on both computation and communication resources. Also, because the event delivery is under the implementation's control and the time needed for the event delivery can be measured and calculated by the implementation, the QoS_requirement can be a deadline for the event delivery; this is not the case for handlers that are generally user-written code. An event receptor's QoS does not have a starting event as part of the QoS_constraint since an event arrival at the receptor is the starting event. Hence, the QoS contains only a QoS_requirement and a timeout for the QoS_constraint.

The QoS_requirement can be of only two forms. One is the deadline, and the other is a priority. A priority is specified by the priority argument. If the QoS_requirement is specified by the deadline, then priority must be the constant MPIRT_PRIORITY_IGNORE. The time_spec is of type MPIRT_TIME_SPEC of the MPIRT_RELATIVE form and represents the deadline when priority is the constant MPIRT_PRIORITY_IGNORE. Otherwise it is a timeout QoS_constraint. ⊤ (final2)

MPIRT_QOS_RECEPTOR_CREATE (name, priority, time_spec, qos_receptor)

IN	name	name of the object (MPIRT_STRING_NAME)
IN	priority	either a priority of an event delivery to the event receptor or the constant MPIRT_PRIORITY_IGNORE for no priority (integer)
IN	time_spec	either a deadline, if priority is MPIRT_PRIORITY_IGNORE, or the timeout or MPIRT_TIMEOUT_IGNORE for no timeout for the delivery of the event to the receptor (MPIRT_TIME_SPEC)
OUT	qos_receptor	event receptor QoS object (ref. to MPIRT_QOS_RECEPTOR)

```
int MPIRT_Qos_receptor_create (MPIRT_String_name name, int priority,
              MPIRT_Time_spec time_spec, MPIRT_Qos_receptor *qos_receptor)
```

```
MPIRT::Qos_Receptor (const MPIRT::String_Name name, int priority,
              MPIRT::Time_Spec time_spec)
```

The next two operations query the parameters of the event receptor QoS object.

MPIRT_QOS_RECEPTOR_GET_PRIORITY (qos_receptor, priority)

IN	qos_receptor	event receptor QoS object (MPIRT_QOS_RECEPTOR)
OUT	priority	either a priority of an event delivery to the event receptor or the constant MPIRT_PRIORITY_IGNORE for no priority (ref. to integer)

```
int MPIRT_Qos_receptor_get_priority (MPIRT_Qos_receptor qos_receptor,
            int *priority)
```

```
int MPIRT::Qos_Receptor::Get_Priority (int &priority) const
```

MPIRT_QOS_RECEPTOR_GET_TIME_SPEC (qos_receptor, time_spec)

IN	qos_receptor	event receptor QoS object (MPIRT_QOS_RECEPTOR)
OUT	time_spec	either a deadline, if priority is MPIRT_PRIORITY_IGNORE, or the timeout for the delivery of the event to the event receptor or MPIRT_TIMEOUT_IGNORE for no timeout (ref. to MPIRT_TIME_SPEC)

```
int MPIRT_Qos_receptor_get_time_spec (MPIRT_Qos_receptor qos_receptor,
            MPIRT_time_spec *time_out)
```

```
int MPIRT::Qos_Receptor::Get_Time_Spec (MPIRT::Time_Spec &time_spec) const
```

The next two operations set the parameters of the event receptor QoS object.

MPIRT_QOS_RECEPTOR_SET_PRIORITY (qos_receptor, priority)

INOUT	qos_receptor	event receptor QoS object (ref. to MPIRT_QOS_RECEPTOR)
IN	priority	either a priority of an event delivery to the event receptor or the constant MPIRT_PRIORITY_IGNORE for no priority (integer)

```
int MPIRT_Qos_receptor_set_priority (MPIRT_Qos_receptor *qos_receptor,
            int priority)
```

```
int MPIRT::Qos_Receptor::Set_Priority (int constraint)
```

MPIRT_QOS_RECEPTOR_SET_TIME_SPEC (qos_receptor, time_spec)

INOUT	qos_receptor	event receptor QoS object (ref. to MPIRT_QOS_RECEPTOR)
IN	time_spec	either a deadline, if priority is MPIRT_PRIORITY_IGNORE, or the timeout for a delivery of the event to the event receptor or MPIRT_TIMEOUT_IGNORE for no timeout (MPIRT_TIME_SPEC)

```
int MPIRT_Qos_receptor_set_time_spec (MPIRT_Qos_receptor *qos_receptor,
              MPIRT_Time_spec time_spec)
```

```
int MPIRT::Qos_Receptor::Set_Time_Spec (MPIRT::Time_Spec time_spec)
```

⊥ (final2)

Duplication, freeing, comparison, and query and set operations for name parameter for the event receptor QoS are derived from the base class MPIRT_OBJECT (see Section 2.2).

11.5 Summary

Miscellaneous Definitions		
Item	*What is it*	*Defined as:*
MPIRT_QOS_NULL	null QOS	null QOS object
MPIRT_QOS_BEST_EFFORT	best-effort QOS	constant of type MPIRT_QOS
MPIRT_QOS_ACCEPT_ANY	accept QOS specification from other channel endpoints	constant of type MPIRT_QOS_CHANNEL
MPIRT_STRING_NAME_NULL	null string name	null (or zero-length) MPIRT_STRING_NAME
MPIRT_TIME_SPEC_IGNORE	best-effort time-spec	null MPIRT_TIME_SPEC
MPIRT_PERIOD_IGNORE	best-effort periodicity	type MPIRT_TIME
MPIRT_TIME_IGNORE	best-effort time	type MPIRT_TIME
MPIRT_PRIORITY_IGNORE	best-effort priority	integer
MPIRT_PRIORITY_INHERIT	inherit receptor priority for a handler	integer
MPIRT_TIMEOUT_IGNORE	best-effort timeout	type MPIRT_TIME_SPEC
MPIRT_BOUND_IGNORE	best-effort event bounds	type MPIRT_TIME
MPIRT_OCCURRENCE_IGNORE	best-effort event occurrence	integer

⊥ (final)

Miscellaneous Bindings		
Item	*C Binding*	*C++ Binding*
MPIRT_QOS_NULL	MPIRT_QOS_NULL	MPIRT::QOS_NULL
MPIRT_QOS_BEST_EFFORT	MPIRT_QOS_BEST_EFFORT	MPIRT::QOS_BEST_EFFORT
MPIRT_QOS_ACCEPT_ANY	MPIRT_QOS_ACCEPT_ANY	MPIRT::QOS_ACCEPT_ANY
MPIRT_TIME_SPEC_IGNORE	MPIRT_TIME_SPEC_IGNORE	MPIRT::TIME_SPEC_IGNORE
MPIRT_PERIOD_IGNORE	MPIRT_PERIOD_IGNORE	MPIRT_PERIOD_IGNORE
MPIRT_TIME_IGNORE	MPIRT_TIME_IGNORE	MPIRT::TIME_IGNORE
MPIRT_PRIORITY_IGNORE	MPIRT_PRIORITY_IGNORE	MPIRT::PRIORITY_IGNORE
MPIRT_PRIORITY_INHERIT	MPIRT_PRIORITY_INHERIT	MPIRT::PRIORITY_INHERIT
MPIRT_TIMEOUT_IGNORE	MPIRT_TIMEOUT_IGNORE	MPIRT::TIMEOUT_IGNORE
MPIRT_BOUND_IGNORE	MPIRT_BOUND_IGNORE	MPIRT::BOUND_IGNORE
MPIRT_OCCURRENCE_IGNORE	MPIRT_OCCURRENCE_IGNORE	MPIRT::OCCURRENCE_IGNORE
MPIRT_QOS_CHANNEL_TIME	MPIRT_Qos_channel_time	MPIRT::Qos_Channel_Time
MPIRT_QOS_CHANNEL_TIME_NULL	MPIRT_QOS_CHANNEL_TIME_NULL	MPIRT::QOS_CHANNEL_TIME_NULL
MPIRT_OBJECT_TYPE_QOS_CHANNEL_TIME	MPIRT_OBJECT_TYPE_QOS_CHANNEL_TIME	MPIRT::OBJECT_TYPE_QOS_CHANNEL_TIME
MPIRT_QOS_CHANNEL_PREV	MPIRT_Qos_channel_prev	MPIRT::Qos_Channel_Prev
MPIRT_QOS_CHANNEL_PREV_NULL	MPIRT_QOS_CHANNEL_PREV_NULL	MPIRT::QOS_CHANNEL_PREV_NULL
MPIRT_OBJECT_TYPE_QOS_CHANNEL_PREV	MPIRT_OBJECT_TYPE_QOS_CHANNEL_PREV	MPIRT::OBJECT_TYPE_QOS_CHANNEL_PREV
MPIRT_QOS_CHANNEL_PRTIEV	MPIRT_Qos_channel_prtiev	MPIRT::Qos_Channel_Prtiev
MPIRT_QOS_CHANNEL_PRTIEV_NULL	MPIRT_QOS_CHANNEL_PRTIEV_NULL	MPIRT::QOS_CHANNEL_PRTIEV_NULL
MPIRT_OBJECT_TYPE_QOS_CHANNEL_PRTIEV	MPIRT_OBJECT_TYPE_QOS_CHANNEL_PRTIEV	MPIRT::OBJECT_TYPE_QOS_CHANNEL_PRTIEV
MPIRT_QOS_HANDLER	MPIRT_Qos_handler	MPIRT::QOS_Handler
MPIRT_QOS_HANDLER_NULL	MPIRT_QOS_HANDLER_NULL	MPIRT::QOS_HANDLER_NULL
MPIRT_OBJECT_TYPE_QOS_HANDLER	MPIRT_OBJECT_TYPE_QOS_HANDLER	MPIRT::OBJECT_TYPE_QOS_HANDLER
MPIRT_QOS_TRIGGER	MPIRT_Qos_trigger	MPIRT::Qos_Trigger
MPIRT_QOS_TRIGGER_NULL	MPIRT_QOS_TRIGGER_NULL	MPIRT::QOS_TRIGGER_NULL
MPIRT_OBJECT_TYPE_QOS_TRIGGER	MPIRT_OBJECT_TYPE_QOS_TRIGGER	MPIRT::OBJECT_TYPE_QOS_TRIGGER
MPIRT_QOS_RECEPTOR	MPIRT_Qos_receptor	MPIRT::Qos_Receptor
MPIRT_QOS_RECEPTOR_NULL	MPIRT_QOS_RECEPTOR_NULL	MPIRT::QOS_RECEPTOR_NULL
MPIRT_OBJECT_TYPE_QOS_RECEPTOR	MPIRT_OBJECT_TYPE_QOS_RECEPTOR	MPIRT::OBJECT_TYPE_QOS_RECEPTOR

Time-Driven Channel QoS Specific Operations		
Language Independent Specification	C	C++
MPIRT_QOS_CHANNEL_TIME_CREATE	MPIRT_Qos_channel_time_create	MPIRT::Qos_Channel_Time
MPIRT_QOS_CHANNEL_TIME_GET_PERIOD	MPIRT_Qos_channel_time_get_period	MPIRT::Qos_Channel_Time::Get_Period
MPIRT_QOS_CHANNEL_TIME_GET_PHASE	MPIRT_Qos_channel_time_get_phase	MPIRT::Qos_Channel_Time::Get_Phase
MPIRT_QOS_CHANNEL_TIME_GET_DEADLINE	MPIRT_Qos_channel_time_get_deadline	MPIRT::Qos_Channel_Time::Get_Deadline
MPIRT_QOS_CHANNEL_TIME_SET_PERIOD	MPIRT_Qos_channel_time_set_period	MPIRT::Qos_Channel_Time::Set_Period
MPIRT_QOS_CHANNEL_TIME_SET_PHASE	MPIRT_Qos_channel_time_set_phase	MPIRT::Qos_Channel_Time::Set_Phase
MPIRT_QOS_CHANNEL_TIME_SET_DEADLINE	MPIRT_Qos_channel_time_set_deadline	MPIRT::Qos_Channel_Time::Set_Deadline
MPIRT_QOS_CHANNEL_TIME_GET_NAME	MPIRT_Qos_channel_time_get_name	MPIRT::Qos_Channel_Time::Get_Name
MPIRT_QOS_CHANNEL_TIME_SET_NAME	MPIRT_Qos_channel_time_set_name	MPIRT::Qos_Channel_Time::Set_Name
MPIRT_QOS_CHANNEL_TIME_DUP	MPIRT_Qos_channel_time_dup	MPIRT::Qos_Channel_Time MPIRT::Qos_Channel_Time::Dup MPIRT::Qos_Channel_Time::Operator=
MPIRT_QOS_CHANNEL_TIME_FREE	MPIRT_Qos_channel_time_time_free	MPIRT::Qos_Channel_∼Time MPIRT::Qos_Channel_Time::Free
MPIRT_QOS_CHANNEL_TIME_IS_EQUAL	MPIRT_Qos_channel_time_is_equal	MPIRT::Qos_Channel_Time::Is_equal
MPIRT_QOS_CHANNEL_TIME_ATTACH_ATTR	MPIRT_Qos_channel_time_attach_attr	MPIRT::Qos_Channel_Time::Attach_Attr
MPIRT_QOS_CHANNEL_TIME_RETRIEVE_ATTR	MPIRT_Qos_channel_time_retrieve_attr	MPIRT::Qos_Channel_Time::Retrieve_Attr
MPIRT_QOS_CHANNEL_TIME_DELETE_ATTR	MPIRT_Qos_channel_time_delete_attr	MPIRT::Qos_Channel_Time::Delete_Attr

Event-Driven With Priority Channel QoS Specific Operations		
Language Independent Specification	C	C++
MPIRT_QOS_CHANNEL_PREV_CREATE	MPIRT_Qos_channel_prev_create	MPIRT::Qos_Channel_Prev
MPIRT_QOS_CHANNEL_PREV_GET_PRIORITY	MPIRT_Qos_channel_prev_get_priority	MPIRT::Qos_Channel_Prev::Get_Priority
MPIRT_QOS_CHANNEL_PREV_GET_START_EVENT	MPIRT_Qos_channel_prev_get_start_event	MPIRT::Qos_Channel_Prev::Get_Start_Event
MPIRT_QOS_CHANNEL_PREV_GET_START_SCOPE	MPIRT_Qos_channel_prev_get_start_scope	MPIRT::Qos_Channel_Prev::Get_Start_Scope
MPIRT_QOS_CHANNEL_PREV_GET_START_PRIORITY	MPIRT_Qos_channel_prev_get_start_priority	MPIRT::Qos_Channel_Prev::Get_Start_Priority
MPIRT_QOS_CHANNEL_PREV_GET_START_TIME_SPEC	MPIRT_Qos_channel_prev_get_start_time_spec	MPIRT::Qos_Channel_Prev::Get_Start_Time_Spec
MPIRT_QOS_CHANNEL_PREV_GET_STOP_EVENT	MPIRT_Qos_channel_prev_get_stop_event	MPIRT::Qos_Channel_Prev::Get_Stop_Event
MPIRT_QOS_CHANNEL_PREV_GET_STOP_SCOPE	MPIRT_Qos_channel_prev_get_stop_scope	MPIRT::Qos_Channel_Prev::Get_Stop_Scope
MPIRT_QOS_CHANNEL_PREV_GET_STOP_PRIORITY	MPIRT_Qos_channel_prev_get_stop_priority	MPIRT::Qos_Channel_Prev::Get_Stop_Priority
MPIRT_QOS_CHANNEL_PREV_GET_STOP_TIME_SPEC	MPIRT_Qos_channel_prev_get_stop_time_spec	MPIRT::Qos_Channel_Prev::Get_Stop_Time_Spec
MPIRT_QOS_CHANNEL_PREV_SET_PRIORITY	MPIRT_Qos_channel_prev_set_priority	MPIRT::Qos_Channel_Prev::Set_Priority
MPIRT_QOS_CHANNEL_PREV_SET_START_EVENT	MPIRT_Qos_channel_prev_set_start_event	MPIRT::Qos_Channel_Prev::Set_Start_Event
MPIRT_QOS_CHANNEL_PREV_SET_START_SCOPE	MPIRT_Qos_channel_prev_set_start_scope	MPIRT::Qos_Channel_Prev::Set_Start_Scope
MPIRT_QOS_CHANNEL_PREV_SET_START_PRIORITY	MPIRT_Qos_channel_prev_set_start_priority	MPIRT::Qos_Channel_Prev::Set_Start_Priority
MPIRT_QOS_CHANNEL_PREV_SET_START_TIME_SPEC	MPIRT_Qos_channel_prev_set_start_time_spec	MPIRT::Qos_Channel_Prev::Set_Start_Time_Spec
MPIRT_QOS_CHANNEL_PREV_SET_STOP_EVENT	MPIRT_Qos_channel_prev_set_stop_event	MPIRT::Qos_Channel_Prev::Set_Stop_Event
MPIRT_QOS_CHANNEL_PREV_SET_STOP_SCOPE	MPIRT_Qos_channel_prev_set_stop_scope	MPIRT::Qos_Channel_Prev::Set_Stop_Scope
MPIRT_QOS_CHANNEL_PREV_SET_STOP_PRIORITY	MPIRT_Qos_channel_prev_set_stop_priority	MPIRT::Qos_Channel_Prev::Set_Stop_Priority
MPIRT_QOS_CHANNEL_PREV_SET_STOP_TIME_SPEC	MPIRT_Qos_channel_prev_set_stop_time_spec	MPIRT::Qos_Channel_Prev::Set_Stop_Time_Spec
MPIRT_QOS_CHANNEL_PREV_GET_NAME	MPIRT_Qos_channel_prev_get_name	MPIRT::Qos_Channel_Prev::Get_Name
MPIRT_QOS_CHANNEL_PREV_SET_NAME	MPIRT_Qos_channel_prev_set_name	MPIRT::Qos_Channel_Prev::Set_Name
MPIRT_QOS_CHANNEL_PREV_DUP	MPIRT_Qos_channel_prev_dup	MPIRT::Qos_Channel_Prev MPIRT::Qos_Channel_Prev::Dup MPIRT::Qos_Channel_Prev::Operator=
MPIRT_QOS_CHANNEL_PREV_FREE	MPIRT_Qos_channel_prev_free	MPIRT::Qos_Channel_~Prev MPIRT::Qos_Channel_Prev::Free
MPIRT_QOS_CHANNEL_PREV_IS_EQUAL	MPIRT_Qos_channel_prev_is_equal	MPIRT::Qos_Channel_Prev::Is_equal
MPIRT_QOS_CHANNEL_PREV_ATTACH_ATTR	MPIRT_Qos_channel_prev_attach_attr	MPIRT::Qos_Channel_Prev::Attach_Attr
MPIRT_QOS_CHANNEL_PREV_RETRIEVE_ATTR	MPIRT_Qos_channel_prev_retrieve_attr	MPIRT::Qos_Channel_Prev::Retrieve_Attr
MPIRT_QOS_CHANNEL_PREV_DELETE_ATTR	MPIRT_Qos_channel_prev_delete_attr	MPIRT::Qos_Channel_Prev::Delete_Attr

Event- and Time-Driven with Priority QoS Specific Operations		
Language Independent Specification	C	C++
MPIRT_QOS_CHANNEL_PRTIEV_CREATE	MPIRT_Qos_channel_prtiev_create	MPIRT::Qos_Channel_Prtiev
MPIRT_QOS_CHANNEL_PRTIEV_GET_PRIORITY	MPIRT_Qos_channel_prtiev_get_priority	MPIRT::Qos_Channel_Prtiev::Get_Priority
MPIRT_QOS_CHANNEL_PRTIEV_GET_START_EVENT	MPIRT_Qos_channel_prtiev_get_start_event	MPIRT::Qos_Channel_Prtiev::Get_Start_Event
MPIRT_QOS_CHANNEL_PRTIEV_GET_START_SCOPE	MPIRT_Qos_channel_prtiev_get_start_scope	MPIRT::Qos_Channel_Prtiev::Get_Start_Scope
MPIRT_QOS_CHANNEL_PRTIEV_GET_START_PRIORITY	MPIRT_Qos_channel_prtiev_get_start_priority	MPIRT::Qos_Channel_Prtiev::Get_Start_Priority
MPIRT_QOS_CHANNEL_PRTIEV_GET_START_TIME_SPEC	MPIRT_Qos_channel_prtiev_get_start_time_spec	MPIRT::Qos_Channel_Prtiev::Get_Start_Time_Spec
MPIRT_QOS_CHANNEL_PRTIEV_GET_STOP_SPEC	MPIRT_Qos_channel_prtiev_get_stop_spec	MPIRT::Qos_Channel_Prtiev::Get_Stop_Spec
MPIRT_QOS_CHANNEL_PRTIEV_SET_PRIORITY	MPIRT_Qos_channel_prtiev_set_priority	MPIRT::Qos_Channel_Prtiev::Set_Priority
MPIRT_QOS_CHANNEL_PRTIEV_SET_START_EVENT	MPIRT_Qos_channel_prtiev_set_start_event	MPIRT::Qos_Channel_Prtiev::Set_Start_Event
MPIRT_QOS_CHANNEL_PRTIEV_SET_START_SCOPE	MPIRT_Qos_channel_prtiev_set_start_scope	MPIRT::Qos_Channel_Prtiev::Set_Start_Scope
MPIRT_QOS_CHANNEL_PRTIEV_SET_START_PRIORITY	MPIRT_Qos_channel_prtiev_set_start_priority	MPIRT::Qos_Channel_Prtiev::Set_Start_Priority
MPIRT_QOS_CHANNEL_PRTIEV_SET_START_TIME_SPEC	MPIRT_Qos_channel_prtiev_set_start_time_spec	MPIRT::Qos_Channel_Prtiev::Set_Start_Time_Spec
MPIRT_QOS_CHANNEL_PRTIEV_SET_STOP_SPEC	MPIRT_Qos_channel_prtiev_set_stop_spec	MPIRT::Qos_Channel_Prtiev::Set_Stop_Spec
MPIRT_QOS_CHANNEL_PRTIEV_GET_NAME	MPIRT_Qos_channel_prtiev_get_name	MPIRT::Qos_Channel_Prtiev::Get_Name
MPIRT_QOS_CHANNEL_PRTIEV_SET_NAME	MPIRT_Qos_channel_prtiev_set_name	MPIRT::Qos_Channel_Prtiev::Set_Name
MPIRT_QOS_CHANNEL_PRTIEV_DUP	MPIRT_Qos_channel_prtiev_dup	MPIRT::Qos_Channel_Prtiev MPIRT::Qos_Channel_Prtiev::Dup MPIRT::Qos_Channel_Prtiev::Operator=
MPIRT_QOS_CHANNEL_PRTIEV_FREE	MPIRT_Qos_channel_prtiev_time_free	MPIRT::Qos_Channel_~Prtiev MPIRT::Qos_Channel_Prtiev::Free
MPIRT_QOS_CHANNEL_PRTIEV_IS_EQUAL	MPIRT_Qos_channel_prtiev_is_equal	MPIRT::Qos_Channel_Prtiev::Is_equal
MPIRT_QOS_CHANNEL_PRTIEV_ATTACH_ATTR	MPIRT_Qos_channel_prtiev_attach_attr	MPIRT::Qos_Channel_Prtiev::Attach_Attr
MPIRT_QOS_CHANNEL_PRTIEV_RETRIEVE_ATTR	MPIRT_Qos_channel_prtiev_retrieve_attr	MPIRT::Qos_Channel_Prtiev::Retrieve_Attr
MPIRT_QOS_CHANNEL_PRTIEV_DELETE_ATTR	MPIRT_Qos_channel_prtiev_delete_attr	MPIRT::Qos_Channel_Prtiev::Delete_Attr

Handler QoS Specific Operations		
Language Independent Specification	C	C++
MPIRT_QOS_HANDLER_CREATE	MPIRT_Qos_handler_create	MPIRT::Qos_Handler
MPIRT_QOS_HANDLER_GET_PRIORITY	MPIRT_Qos_handler_get_priority	MPIRT::Qos_Handler::Get_Priority
MPIRT_QOS_HANDLER_GET_TIMEOUT	MPIRT_Qos_handler_get_timeout	MPIRT::Qos_Handler::Get_Timeout
MPIRT_QOS_HANDLER_SET_PRIORITY	MPIRT_Qos_handler_set_priority	MPIRT::Qos_Handler::Set_Priority
MPIRT_QOS_HANDLER_SET_TIMEOUT	MPIRT_Qos_handler_set_timeout	MPIRT::Qos_Handler::Set_Timeout
MPIRT_QOS_HANDLER_GET_NAME	MPIRT_Qos_handler_get_name	MPIRT::Qos_Handler::Get_Name
MPIRT_QOS_HANDLER_SET_NAME	MPIRT_Qos_handler_set_name	MPIRT::Qos_Handler::Set_Name
MPIRT_QOS_HANDLER_DUP	MPIRT_Qos_handler_dup	MPIRT::Qos_Handler MPIRT::Qos_Handler::Dup MPIRT::Qos_Handler::Operator=
MPIRT_QOS_HANDLER_FREE	MPIRT_Qos_handler_time_free	MPIRT::Qos::∼Handler MPIRT::Qos_Handler::Free
MPIRT_QOS_HANDLER_IS_EQUAL	MPIRT_Qos_handler_is_equal	MPIRT::Qos_Handler::Is_equal
MPIRT_QOS_HANDLER_ATTACH_ATTR	MPIRT_Qos_handler_attach_attr	MPIRT::Qos_Handler::Attach_Attr
MPIRT_QOS_HANDLER_RETRIEVE_ATTR	MPIRT_Qos_handler_retrieve_attr	MPIRT::Qos_Handler::Retrieve_Attr
MPIRT_QOS_HANDLER_DELETE_ATTR	MPIRT_Qos_handler_delete_attr	MPIRT::Qos_Handler::Delete_Attr

Trigger QoS Specific Operations		
Language Independent Specification	C	C++
MPIRT_QOS_TRIGGER_CREATE	MPIRT_Qos_trigger_create	MPIRT::Qos::Trigger
MPIRT_QOS_TRIGGER_GET_BOUND	MPIRT_Qos_trigger_get_bound	MPIRT::Qos::Trigger::Get_Bound
MPIRT_QOS_TRIGGER_GET_PERIOD	MPIRT_Qos_trigger_get_period	MPIRT::Qos::Trigger::Get_Period
MPIRT_QOS_TRIGGER_GET_OCCURRENCE	MPIRT_Qos_trigger_get_occurrence	MPIRT::Qos::Trigger::Get_Occurrence
MPIRT_QOS_TRIGGER_SET_BOUND	MPIRT_Qos_trigger_set_bound	MPIRT::Qos::Trigger::Set_Bound
MPIRT_QOS_TRIGGER_SET_PERIOD	MPIRT_Qos_trigger_set_period	MPIRT::Qos::Trigger::Set_Period
MPIRT_QOS_TRIGGER_SET_OCCURRENCE	MPIRT_Qos_trigger_set_occurrence	MPIRT::Qos::Trigger::Set_Occurrence
MPIRT_QOS_TRIGGER_GET_NAME	MPIRT_Qos_trigger_get_name	MPIRT::Qos::Trigger::Get_Name
MPIRT_QOS_TRIGGER_SET_NAME	MPIRT_Qos_trigger_set_name	MPIRT::Qos::Trigger::Set_Name
MPIRT_QOS_TRIGGER_DUP	MPIRT_Qos_trigger_dup	MPIRT::Qos::Trigger MPIRT::Qos::Trigger::Dup MPIRT::Qos::Trigger::Operator=
MPIRT_QOS_TRIGGER_FREE	MPIRT_Qos_trigger_time_free	MPIRT::Qos::∼Trigger MPIRT::Qos::Trigger::Free
MPIRT_QOS_TRIGGER_IS_EQUAL	MPIRT_Qos_trigger_is_equal	MPIRT::Qos::Trigger::Is_equal
MPIRT_QOS_TRIGGER_ATTACH_ATTR	MPIRT_Qos_trigger_attach_attr	MPIRT::Qos::Trigger::Attach_Attr
MPIRT_QOS_TRIGGER_RETRIEVE_ATTR	MPIRT_Qos_trigger_retrieve_attr	MPIRT::Qos::Trigger::Retrieve_Attr
MPIRT_QOS_TRIGGER_DELETE_ATTR	MPIRT_Qos_trigger_delete_attr	MPIRT::Qos::Trigger::Delete_Attr

Receptor QoS Specific Operations		
Language Independent Specification	C	C++
MPIRT_QOS_RECEPTOR_CREATE	MPIRT_Qos_receptor_create	MPIRT::Qos_Receptor
MPIRT_QOS_RECEPTOR_GET_PRIORITY	MPIRT_Qos_receptor_get_priority	MPIRT::Qos_Receptor::Get_Priority
MPIRT_QOS_RECEPTOR_GET_TIME_SPEC	MPIRT_Qos_receptor_get_time_spec	MPIRT::Qos_Receptor::Get_Time_Spec
MPIRT_QOS_RECEPTOR_SET_PRIORITY	MPIRT_Qos_receptor_set_priority	MPIRT::Qos_Receptor::Set_Priority
MPIRT_QOS_RECEPTOR_SET_TIME_SPEC	MPIRT_Qos_receptor_set_time_spec	MPIRT::Qos_Receptor::Set_Time_Spec
MPIRT_QOS_RECEPTOR_GET_NAME	MPIRT_Qos_receptor_get_name	MPIRT::Qos_Receptor::Get_Name
MPIRT_QOS_RECEPTOR_SET_NAME	MPIRT_Qos_receptor_set_name	MPIRT::Qos_Receptor::Set_Name
MPIRT_QOS_RECEPTOR_DUP	MPIRT_Qos_receptor_dup	MPIRT::Qos_Receptor MPIRT::Qos_Receptor::Dup MPIRT::Qos_Receptor::Operator=
MPIRT_QOS_RECEPTOR_FREE	MPIRT_Qos_receptor_time_free	MPIRT::Qos::∼Receptor MPIRT::Qos_Receptor::Free
MPIRT_QOS_RECEPTOR_IS_EQUAL	MPIRT_Qos_receptor_is_equal	MPIRT::Qos_Receptor::Is_equal
MPIRT_QOS_RECEPTOR_ATTACH_ATTR	MPIRT_Qos_receptor_attach_attr	MPIRT::Qos_Receptor::Attach_Attr
MPIRT_QOS_RECEPTOR_RETRIEVE_ATTR	MPIRT_Qos_receptor_retrieve_attr	MPIRT::Qos_Receptor::Retrieve_Attr
MPIRT_QOS_RECEPTOR_DELETE_ATTR	MPIRT_Qos_receptor_delete_attr	MPIRT::Qos_Receptor::Delete_Attr

Chapter 12

Committing Objects and Resource Allocation

Current Status: First formal vote passed 11/0/0 on 6/23/98.
Second formal vote passed 8/0/2 in July 98.
Third formal vote passed 9/0/0 on 12/3/98.

This chapter defines the second phase of the deferred early binding of the life cycle of an object. The commit phase supports the commitment of deferred early binding initialization of the objects. This phase does the admission test of the objects. If the admission test passes, then resources are allocated for these objects, and they become committed.

12.1 Commit Operation

The admission test rigorously validates all objects in the supplied container. Once the objects are committed, the application is not allowed to change them. It is a user responsibility to ensure that there are no operations pending on the objects that are in the designated container when the MPIRT_COMMIT is called.

The following operation is collective and is synchronized over the group MPIRT_GROUP_WORLD.

MPIRT_COMMIT (container)

INOUT	container	a container that includes all objects to be committed (ref. to MPIRT_CSET)

```
int MPIRT_Commit (MPIRT_Cset *container)
```

```
int MPIRT::Commit (MPIRT::Cset &container)
```

Rationale. The above function performs a collective admission test of a set of objects previously constructed and returns the status of the admission process. If the test is successful, the above operation transitions the objects in the container into the committed state and allocates resources for them. (*End of rationale.*)

The return code for this operation is one of the following:

MPIRT_SUCCESS The admission test is successful, the resources are allocated for all the objects in the container, and the application has transitioned into the real-time mode with the container objects committed.

MPIRT_ERR_INSUF_MEMORY There is not enough memory to satisfy the request.

MPIRT_ERR_INSUF_NETWORK There are insufficient network resources to satisfy the request. This error means that either there are insufficient network resources to satisfy the quality-of-service (QoS) requested or that the network cannot support the requested channel message transfer or event delivery requirements.

MPIRT_ERR_SCHEDULE There are insufficient resources to satisfy the requested object's QoS schedules.

MPIRT_ERR_ILLEGAL_BUFFER A buffer presented in the initial_buffer_list is not in allowed_buffer_list for a buffer iterator in the container.

MPIRT_ERR_NOT_A_BUFFER An object provided for an initial_buffer_list or allowed_buffer_list for a buffer iterator in the container is not a buffer.

MPIRT_ERR_DATASPEC_MISMATCH A dataspec or a bufsize parameters for a buffer in the initial_buffer_list or in the allowed_buffer_list does not match these parameters for a buffer iterator in the container.

MPIRT_ERR_NOT_MAPPED A policies of the buffer iterators for a channel end points do not match, or the number of buffers, or their dataspecs, or their buffer sizes for channel end points is different that is not allowed for the buffer iterator policy.

MPIRT_ERR_INV_OBJECT An invalid object in the container is not valid. For example, a receptor with event name MPIRT_STRING_NAME_NULL is in the container.

MPIRT_COMMIT preserves all the previously committed objects that also appear in the designated container, and maintains the state these objects were in before the operation. However, it is assumed that there are no pending transfers in progress or other pending operations on these objects when the commit operation is called. There is no incrementally committing objects from the designated container; either all the objects are committed or none are committed.

If the designated container contains other containers MPIRT_COMMIT traverses and commits committable objects in all nested containers. If an object appears more than once in the designated container or in its nested containers it will be committed only once. An implementation not only commits all the committable objects in the designated container and its nested containers but also all their constituted objects. That means that if a channel appears in the designated container then its buffer iterators and buffers that appear in theirs allowed buffer lists and initial buffer lists will also be committed. Non committable objects are allowed to appear in the designated container or in the containers nested inside and the MPIRT_COMMIT operation does not have any effect on them.

The MPIRT_COMMIT operation is responsible for inserting the buffers from the initial_buffer_list into the buffer iterator object if the MPIRT_COMMIT is successful.

Figure 12.1 shows that the commit operation changes the state of the previously committed objects that do not appear in the designated container into the MPIRT_*name-of-class*_UNCOMMITTED state while preserving the MPIRT_*name-of-class*_COMMITTED state (one of several states of the committed channel) of the other previously committed objects. Other objects from the designated

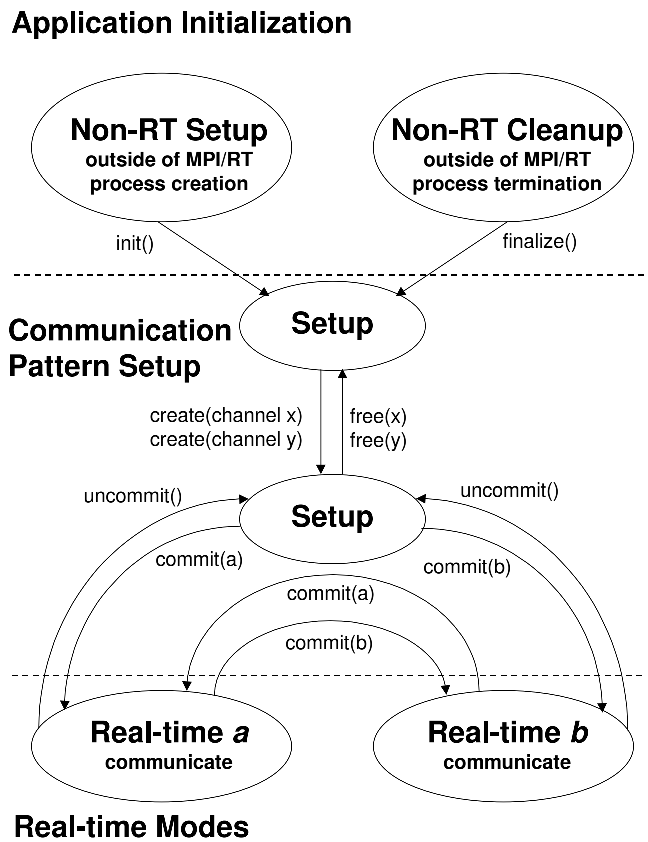

Figure 12.1: A basic real-time program has four phases: startup (non-real-time), setup (non-real-time), real-time mode(s), and cleanup (non-real-time).

container that were in the MPIRT_*name-of-class*_UNCOMMITTED state prior to the MPIRT_COMMIT operation transition into the MPIRT_*name-of-class*_COMMITTED state (MPIRT_CHANNEL_IDLE for channels). If the program was in real-time mode, the MPIRT_COMMIT operation temporarily leaves real-time phase and reenters the communication setup phase while it performs the admission test for the new set of objects. ⊥ (final2)

The following operation is collective and is synchronized over the group MPIRT_GROUP_WORLD. It allows users to uncommit objects that are committed. While MPIRT_COMMIT allows the user to commit resources and transition from one committed set of objects to the next, MPIRT_UNCOMMIT allows users to uncommit all currently committed objects and reclaim all their resources. This operation is required prior to the free operation for a committable object. ⊥ (final)

MPIRT_UNCOMMIT()

```
int MPIRT_Uncommit(void)
```

`int MPIRT::Uncommit (void)`

Rationale. This operation is equivalent to MPIRT_COMMIT with an empty container or with MPIRT_CONTAINER_NULL and with an additional caveat that all resources must be reclaimed by the implementation. (*End of rationale.*)

12.2 Summary

This section summarizes enumerations and type declarations that are introduced in this chapter.

Admission Test Return Codes	
Error Code	*meaning*
MPIRT_SUCCESS	test successful
MPIRT_ERR_INSUF_MEMORY	failure due to lack of memory
MPIRT_ERR_INSUF_NETWORK	failure due to insufficient network resources
MPIRT_ERR_SCHEDULE	failure, unable to guarantee QoS schedules
MPIRT_ERR_ILLEGAL_BUFFER	illegal buffer in the allowed_buffer_list or in the initial_buffer_list for some buffer iterator
MPIRT_ERR_NOT_A_BUFFER	not a buffer object is in the allowed_buffer_list or in the initial_buffer_list for some buffer iterator
MPIRT_ERR_DATASPEC_MISMATCH	the data descriptors mismatched between a buffer iterator and one of its buffers
MPIRT_ERR_NOT_MAPPED	the policies or data descriptors between a channel endpoints buffer iterators

Miscellaneous Bindings		
Item	*C Bindings*	*C++ Bindings*
MPIRT_ERR_INSUF_MEMORY	MPIRT_ERR_INSUF_MEMORY	MPIRT::ERR_INSUF_MEMORY
MPIRT_ERR_INSUF_NETWORK	MPIRT_ERR_INSUF_NETWORK	MPIRT::ERR_INSUF_NETWORK
MPIRT_ERR_SCHEDULE	MPIRT_ERR_SCHEDULE	MPIRT::ERR_SCHEDULE
MPIRT_ERR_ILLEGAL_BUFFER	MPIRT_ERR_ILLEGAL_BUFFER	MPIRT::ERR_ILLEGAL_BUFFER
MPIRT_ERR_NOT_A_BUFFER	MPIRT_ERR_NOT_A_BUFFER	MPIRT::ERR_NOT_A_BUFFER
MPIRT_ERR_DATASPEC_MISMATCH	MPIRT_ERR_DATASPEC_MISMATCH	MPIRT::ERR_DATASPEC_MISMATCH
MPIRT_ERR_NOT_MAPPED	MPIRT_ERR_NOT_MAPPED	MPIRT::ERR_NOT_MAPPED

Addmission Test Operations		
Language Independent Specification	C	C++
MPIRT_COMMIT	MPIRT_Commit	MPIRT::Commit
MPIRT_UNCOMMIT	MPIRT_Uncommit	MPIRT::Uncommit

Part IV

Environmental Mechanisms and Functionality

Chapter 13

Initialization and Termination

Current Status: First formal vote passed 9/0/0 on 4/10/98.
Second formal vote passed 9/0/0 on 6/25/98.
Third formal vote passed 9/0/0 on 12/3/98.

13.1 Initialization and Termination of MPI/RT

The primary goal of MPI/RT is to achieve source code portability. That means that the program, utilizing a relevant language's standard and that language's MPI/RT bindings, must not require any source code changes when moved from one system to another. The issues concerning *performance portability* are not directly related to initialization and termination and are discussed separately in Chapter 1. MPI/RT does not specify how MPI/RT programs start or terminate nor what users need to do to set up an environment in which an MPI/RT program will run. However, an implementation may require some setup to be performed before other MPI/RT operations can be called. This is accomplished with the operation MPIRT_INIT. Initialization of the MPI/RT environment needs to be flexible enough to support a mixture of processes requiring real-time features with those that may not.

Arguments can be passed from the startup mechanism to the MPI/RT library through MPIRT_INIT using argc and argv.

MPIRT_INIT(argc,argv)

INOUT	argc	pointer to the number of arguments in the argument vector (integer)
INOUT	argv	pointer to the argument vector (pointer to array of strings)

```
int  MPIRT_Init (int *argc, char **argv[])
```

```
int MPIRT::Init (int &argc, char &**argv[])
```

Advice to implementors. Interpretation of the parameters to MPIRT_INIT is implementation specific. (*End of advice to implementors.*)

Advice to users. The parameters argc and argv may be changed by MPIRT_INIT in order to add or delete parameters for implementation-specific reasons. (*End of advice to users.*)

This routine must be called before any other MPI/RT routine (except MPIRT_IS_INITIALIZED or MPIRT_IS_FINALIZED). All MPI/RT programs must contain a call to MPIRT_INIT. It must be called exactly once, and all subsequent calls are erroneous.

The creation of the group MPIRT_GROUP_WORLD and the synchronized clock across all processes is a requirement for the completion of MPIRT_INIT. Upon completion of MPIRT_INIT, MPI/RT functionality is ready to be used. This function may return the values MPIRT_SUCCESS, MPIRT_FAILURE, or MPIRT_ERR_TIMEOUT.

Advice to implementors. If a timeout is supported, then the method of specification is implementation-defined. For example, the timeout value may be specified as a command-line argument, an environment variable, or a constant. (*End of advice to implementors.*)

MPIRT_INIT does NOT create any application processes. No new processes that will have a rank in MPIRT_GROUP_WORLD can be created by MPIRT_INIT. Processes are created by mechanisms outside of this standard. This call is considered to be collective in that all participating processes are required to call it, but it does not necessarily imply synchronization of processes inside the call.

Advice to users. Users should not rely on an implementation's use of MPIRT_INIT as a synchronization point. The function MPIRT_COMMIT (see Chapter 12) is a collective, synchronized call that must be made prior to any application communication. (*End of advice to users.*)

Advice to implementors. The implementation may use MPIRT_INIT as a synchronization point, but it is not required to be one. The implementation is free to support this call as a local call. (*End of advice to implementors.*)

MPIRT_FINALIZE()

```
int MPIRT_Finalize(void)
```

```
int MPIRT::Finalize (void)
```

This routine cleans up all MPI/RT states. Once called, no other MPI/RT routines (except MPIRT_IS_INITIALIZED or MPIRT_IS_FINALIZED) may be invoked. The application must ensure that all pending communication has completed prior to calling MPIRT_FINALIZE. This function may return MPIRT_SUCCESS or MPIRT_FAILURE.

MPIRT_FINALIZE does NOT destroy any application processes. No processes that had a rank in MPIRT_GROUP_WORLD can be terminated by this operation. Processes are destroyed by mechanisms outside of this standard. This function is considered to be collective in that all participating processes are required to call it, but it does not necessarily imply synchronization of processes inside the call.

Advice to users. Users should not rely on an implementation's use of MPIRT_FINALIZE as a synchronization point. The function MPIRT_UNCOMMIT (see Chapter 12), which frees MPI/RT resources used by the process, is a collective, synchronized call that must be made prior to MPIRT_FINALIZE. (*End of advice to users.*)

Advice to implementors. The implementation may use MPIRT_FINALIZE as a synchronization point, but it is not required to be one. The implementation is free to support this call as a local call. (*End of advice to implementors.*)

MPIRT_IS_INITIALIZED(flag)

OUT	flag	flag is MPIRT_TRUE if MPIRT_INIT has been called and MPIRT_FALSE otherwise (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Is_initialized (MPIRT_Boolean *flag)
```

```
int MPIRT::Is_Initialized (MPIRT::Boolean &flag) const
```

This routine returns MPIRT_TRUE if and only if MPIRT_INIT has been called and MPIRT_FALSE otherwise. This function is one of the two functions that may be called before MPIRT_INIT or after MPIRT_FINALIZE. The behavior of MPIRT_IS_INITIALIZED is not affected by the call of MPIRT_FINALIZE.

MPIRT_IS_FINALIZED(flag)

OUT	flag	flag is MPIRT_TRUE if MPIRT_FINALIZE has been called and MPIRT_FALSE otherwise (ref. to MPIRT_BOOLEAN)

```
int MPIRT_Is_finalized (MPIRT_Boolean *flag)
```

```
int MPIRT::Is_Finalized (MPIRT::Boolean &flag) const
```

This routine returns MPIRT_TRUE if and only if MPIRT_FINALIZE has completed. Otherwise, it returns MPIRT_FALSE. This function is one of the two functions that may be called before MPIRT_INIT or after MPIRT_FINALIZE.

Rationale. This operation allows library writers to detect that MPI/RT is no longer active. (*End of rationale.*)

Figure 12.1 shows the phases of an MPI/RT program.

13.2 Version Information

Header files for MPI/RT must include constant declarations for compile time determination of the version and subversion of the standard that is implemented by the libraries. These constants are called MPIRT_VERSION and MPIRT_SUBVERSION. In C and C++, for instance, these values are declared by using the preprocessor directive `#define`. For the version of MPI/RT specified by this document, the value of MPIRT_VERSION is the integer *1*, and MPIRT_SUBVERSION is the integer *0*.

To facilitate run-time checking of the version implemented by the library, the following function is defined.

MPIRT_SPEC_VERSION (version, subversion)

OUT	version	the major version number of the standard supported by the library (ref. to integer)
OUT	subversion	the minor version number of the standard supported by the library (ref. to integer)

```
int MPIRT_Spec_version (int *version, int *subversion)
```

```
int MPIRT::Spec_Version (int &version, int &subversion) const
```

Rationale. Both compile time and runtime methods of determining version information are needed to check that libraries linked into the application implement functionality expected by the program and that included implementation files are of the appropriate version. The runtime version is supplied to facilitate this functionality in the face of shared libraries which are not linked until runtime. (*End of rationale.*)

Advice to users. It may be possible to write applications that will work with libraries implementing earlier versions of the standard. (*End of advice to users.*)

Advice to implementors. As with all functions that return constant values, this function may be implemented as an inline function. (*End of advice to implementors.*)

13.3 Summary

This section summarizes enumerations and type definitions that are introduced in this chapter.

Miscellaneous Definitions		
Definition	*What is it*	*How defined?*
MPIRT_GROUP_WORLD	predefined group of all participating processes	implementation
MPIRT_VERSION	major version number of the specification supported by the library	integer value 1
MPIRT_SUBVERSION	minor version number of the specification supported by the library	integer value 0

Initialization Error Codes	
Error Code	*Returned When*
MPIRT_SUCCESS	initialization successful
MPIRT_FAILURE	general failure
MPIRT_ERR_TIMEOUT	initialization timeout

Miscellaneous Bindings		
Item	*C Bindings*	*C++ Bindings*
MPIRT_VERSION	MPIRT_VERSION	MPIRT::VERSION
MPIRT_SUBVERSION	MPIRT_SUBVERSION	MPIRT::SUBVERSION
MPIRT_ERR_TIMEOUT	MPIRT_ERR_TIMEOUT	MPIRT::ERR_TIMEOUT

Initialization and Termination Operations		
Language Independent Specification	C	C++
MPIRT_INIT	MPIRT_Init	MPIRT::Init
MPIRT_FINALIZE	MPIRT_Finalize	MPIRT::Finalize
MPIRT_IS_INITIALIZED	MPIRT_Is_initialized	MPIRT::Is_Initialized
MPIRT_IS_FINALIZED	MPIRT_Is_finalized	MPIRT::Is_Finalized
MPIRT_SPEC_VERSION	MPIRT_Spec_version	MPIRT::Spec_Version

Chapter 14

Clocks

Current Status: First formal vote passed 11/0/0 on 16-Jan-98.
Second formal vote passed 7/0/0 on 24-Feb-98.
Third formal vote passed 9/0/0 on 21/3/98.

14.1 Synchronization of Clocks

Most platforms that support real-time applications provide tightly synchronized system clocks that are dependent upon special hardware support. There are several compelling reasons for having tightly synchronized clocks [12]:

1. Fine-grained, accurate instrumentation is needed for all approaches to real-time message-passing systems and even for performance measurements in non-real-time systems.

2. Real-time applications require precise timing correctness. These systems also require demonstrations of this correctness and often require delicate tuning for optimal performance. Tightly synchronized clocks are necessary to support these requirements in a parallel environment.

3. Applications should be able to adjust scheduled times for portability, based upon the quality of the synchronized clocks. For example, time padding will be needed to adapt scheduled message-passing operations on heterogeneous processing nodes.

4. The primary goal of time-driven, real-time MPI is to support application specification of resource use. For time-driven MPI/RT, all resources that are used for communication need to be scheduled in order to achieve predictable behavior. These scheduled resources can include the following:

   (a) Distributed Memory,
   (b) Shared Memory,
   (c) Communication Bandwidth,
   (d) Communication Fabric,
   (e) Computation, and
   (f) Others, such as platform-specific resources related to inter-process communication.

14.2 Description of the Clocks

MPI/RT provides a *synchronized clock* service. The processing nodes (which may contain one or more CPUs) that comprise a real-time parallel processing system may have access to several clocks. One of the clocks is designated as a "global master clock." Synchronization of the master clock to an external time reference is left up to the implementation.

> *Advice to implementors.* MPI/RT implementations can use the platform-provided synchronized clock, if it is available, or else provide synchronization of clocks as part of its services.
>
> For most platforms, each node has a local clock service which is periodically corrected in order for it to serve as a synchronized clock (with respect to the master clock) for all MPI/RT processes active at that node. However, other alternatives are not precluded. For example, there may be a single clock at one of the nodes or one completely outside of the participating nodes, which all of the processing nodes access over a network (which may also be used for regular data transfer operations) to get a synchronized clock value. (*End of advice to implementors.*)

For each process in an MPI/RT program, the synchronized clock will be the one read by the MPIRT_CLOCK_TIME call. There is an underlying assumption that each process is associated with a fixed processing node. The process accesses its globally synchronized clock through its associated node.

MPI/RT's synchronized clock is guaranteed to be monotonically non-decreasing.

> *Advice to implementors.* The underlying middleware and/or runtime system and/or operating system should hide any artifacts resulting from the overflow of system clock counters. (*End of advice to implementors.*)

14.3 Clock Synchronization Parameters

This section presents several parameters that describe the synchronized clock, all of which must be accessible at runtime. The values of all of these parameters are expected to be of type MPIRT_TIME in units of nanoseconds, with the exception of drift, which is a dimensionless quantity.

The values of absolute time are relative to the MPI/RT *epoch*, which is implementation-defined.

14.3.1 The Epoch

MPI/RT provides a quantity called the MPI/RT *epoch offset* which measures, in nanoseconds, the difference between the MPI/RT epoch and the *POSIX epoch* that would be returned on the local node. A positive value for this constant indicates that the MPI/RT value is in the future with regard to the POSIX epoch [9]. The function MPIRT_CLOCK_GET_EPOCH_OFFSET returns the MPI/RT epoch offset of type MPIRT_TIME.

> *Rationale.* It is important to allow implementors to exploit POSIX, if available, without requiring the use of the POSIX-defined epoch. However, connecting multiple implementations with distinct, fixed epochs could cause an interoperability impasse. (*End of rationale.*)

MPIRT_CLOCK_GET_EPOCH_OFFSET (epoch_offset)

OUT	epoch_offset	difference between the MPI/RT epoch and the *POSIX* epoch, measured in nanoseconds (ref. to MPIRT_TIME)

```
int MPIRT_Clock_get_epoch_offset (MPIRT_Time *epoch_offset)
```

```
int MPIRT::Clock::Get_Epoch_Offset (MPIRT::Time &epoch_offset)
```

14.3.2 The MPIRT_TIME Type

While it is formally defined in Chapter 3, we recall here that MPIRT_TIME is defined as MPIRT_INT64 (see Chapter 3).

14.3.3 The Synchronized Time Service

The following is the synchronized clock service of MPI/RT.

MPIRT_CLOCK_TIME (time)

OUT	time	the time since the MPI/RT epoch, measured in nanoseconds (ref. to MPIRT_TIME)

```
int MPIRT_Clock_time (MPIRT_Time *time)
```

```
int MPIRT::Clock::Time (MPIRT::Time &time)
```

14.3.4 Parameters

The following quantities describe the quality of the synchronized clock. In what follows, the MPI/RT implementation returns MPIRT_CLOCK_TIME_UNDEFINED for accuracy, skew, and access time whenever it has no established quality-of-service (QoS) for these quantities.

Resolution (Tick) Resolution represents the time between two successive clock ticks. The resolutions of the various synchronized clocks in a heterogeneous system may differ.

> *Advice to implementors.* The MPIRT_CLOCK_TICK call returns the resolution of the synchronized clock of the processing node associated with the calling process. (*End of advice to implementors.*)

MPIRT_CLOCK_TICK (tick)

OUT	tick	a measure of the time between two successive clock ticks, measured in nanoseconds (ref. to MPIRT_TIME)

```
int MPIRT_Clock_tick (MPIRT_Time *tick)
```

```
int MPIRT::Clock::Tick (MPIRT::Time &tick)
```

Drift Drift indicates, for the synchronized clock, a guaranteed upper bound on the error in the rate of the clock as compared to the master clock. Although an upper bound, drift may be a dynamically updated quantity during system execution.

By definition, the master clock has zero drift. That is, if the drift is $\delta < 1$, then the clock rate (measured in seconds per actually elapsed second) is guaranteed to be between $1 - \delta$ and $1 + \delta$. Drift is, as stated above, dimensionless.

Drift can be effectively used to bound the accuracy of measurement of small time intervals when they are measured by the difference between two readings of the same synchronized clock.

Rationale. Inasmuch as it is desired to avoid mandating the floating point representation of drift, a scaled drift is returned as an integer quantity, namely $2^{32}\delta$. This allows users to capture resolutions below one nanosecond while retaining the simplicity of integer representation and allows simple shifting to put delta back into a fixed-point form. (*End of rationale.*)

Advice to implementors. An implementation of MPI/RT should recalculate the drift parameter when global system clocks are synchronized. (*End of advice to implementors.*)

MPIRT_CLOCK_DRIFT (drift)

OUT	drift	measured as shifted dimensionless quantity of error of the rate of the synchronized clock (ref. to MPIRT_INT64)

```
int  MPIRT_Clock_Drift (MPIRT_Int64 *drift)
```

```
int MPIRT::Clock::Drift (MPIRT::Int64 &drift)
```

Skew Skew is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clocks in distinct nodes. Note that this refers to ideal values, not the result of any real reading operations.

If the skew is undetermined, an implementation returns MPIRT_CLOCK_TIME_UNDEFINED for skew.

MPIRT_CLOCK_SKEW (skew)

OUT	skew	maximum bound on the difference between pairs of synchronized clocks over the system (ref. to MPIRT_TIME)

```
int  MPIRT_Clock_skew (MPIRT_time *skew)
```

```
int MPIRT::Clock::Skew (MPIRT::Time &skew)
```

Accuracy Accuracy is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clock and an ideal clock started at the synchronized clock's hypothetical starting time (some critical instant in time).

If the system is not synchronized to the external time reference, an implementation returns MPIRT_CLOCK_TIME_UNDEFINED for accuracy.

Advice to users. If synchronized clocks are periodically corrected in order to deal with drifts and other inaccuracies, the calculation of the interval accuracy based upon drift will be too pessimistic for large intervals. (*End of advice to users.*)

This value can also be used for cross-platform synchronization when implementations do not provide this interoperability.

MPIRT_CLOCK_ACCURACY (accuracy)

OUT	accuracy	maximum bound on the time difference between the synchronized clock and an ideal clock (ref. to MPIRT_TIME)

```
int  MPIRT_Clock_accuracy (MPIRT_Time *accuracy)
```

```
int MPIRT::Clock::Accuracy (MPIRT::Time &accuracy)
```

Access Time Access time is a maximum bound on the time to execute a call of MPIRT_CLOCK_TIME on an idle system.

Rationale. The time to execute the call may be longer if it is interrupted by the operating system. This undesirable and ambiguous caveat is necessary with current operating systems (especially those with virtual memory) because an absolute guarantee would be so long as to be practically unusable. (*End of rationale.*)

If the access time bound is unknown, then an implementation returns MPIRT_CLOCK_TIME_UNDEFINED for the access time.

Rationale. For example, implementations that are based on accessing the same global clock instead of a local synchronized one will have the same resolution for all nodes, with zero skew, no drift, and identical accuracy (subject to the propagation delay of the clock signal) for all processing nodes. However, the access times could still vary at each processing node. (*End of rationale.*)

MPIRT_CLOCK_ACCESS_TIME (access_time)

OUT	access_time	maximum bound on the execution time of MPIRT_CLOCK_TIME (ref. to MPIRT_TIME)

```
int  MPIRT_Clock_access_time (MPIRT_Time *accuracy)
```

```
int MPIRT::Clock::Access_Time (MPIRT::Time &access_time)
```

14.4 Behavior of the Time Services

The possible return codes are as follows:

1. MPIRT_SUCCESS
2. MPIRT_FAILURE

When MPIRT_FAILURE is returned, the return value for a time value is MPIRT_CLOCK_TIME_UNDEFINED.

14.5 Summary

Miscellaneous Definitions		
Item	*What is it*	*Defined as:*
MPIRT_CLOCK_TIME_UNDEFINED	undefined time value	value of type MPIRT_TIME

Miscellaneous Bindings		
Item	*C Bindings*	*C++ Bindings*
MPIRT_CLOCK_TIME_UNDEFINED	MPIRT_CLOCK_TIME_UNDEFINED	MPIRT::Clock::TIME_UNDEFINED

Clock Operations		
Language Independent Specification	C	C++
MPIRT_CLOCK_GET_EPOCH_OFFSET	MPIRT_Clock_get_epoch_offset	MPIRT::Clock::Get_Epoch_Offset
MPIRT_CLOCK_TIME	MPIRT_Clock_time	MPIRT::Clock::Time
MPIRT_CLOCK_TICK	MPIRT_Clock_tick	MPIRT::Clock::Tick
MPIRT_CLOCK_DRIFT	MPIRT_Clock_drift	MPIRT::Clock::Drift
MPIRT_CLOCK_SKEW	MPIRT_Clock_skew	MPIRT::Clock::Skew
MPIRT_CLOCK_ACCURACY	MPIRT_Clock_accuracy	MPIRT::Clock::Accuracy
MPIRT_CLOCK_ACCESS_TIME	MPIRT_Clock_access_time	MPIRT::Clock::Access_Time

Chapter 15

Instrumentation

Current Status: First formal vote passed 9/0/2 in June 98.
Second formal vote passed 6/1/3 in July 98.
Third formal vote passed 8/0/1 on 12/3/98.

⊤ (final2)

15.1 Introduction

Instrumentation is an essential aspect in providing application developers with the tools needed to monitor quality-of-service (QoS) assurances and fine-tune specific platform configurations [10]. Performance portability, maintainability, and fault tolerance can be enhanced using instrumentation. For real-time instrumentation is critical to the performance tuning of the application. ⊤ (final)

Real-time instrumentation must be flexible and minimize any impact on the timing behavior ⊥ (final) of the application and its communication. MPI/RT instrumentation makes available a variety of metrics, but does not specify how this information is used. It is up to the application to log the information or otherwise process it. This design promotes implementation independence. Possible uses of this information include the following:

- Profiling MPI/RT, perhaps in conjunction with a profiling tool, by simply printing the probe information.
- Real-time performance tuning by analyzing the probe information and modifying the behavior of the application.
- Fault detection by analyzing the probe information looking for values that are not changing or are getting out of tolerance.

In MPI/RT, a *metric* is typically a scalar value that is maintained by the system. Metrics keep track of object-specific information such as I/O counts in channels, as well as system-wide information such as elapsed time. Some metrics are required to be implemented by a conforming MPI/RT implementation, but additional metrics may be defined by the implementation. All of the standard metrics are monotonically increasing.

Metrics are not accessed directly by the application. Instead, software *probes* are attached to the metrics. This hides the actual location of the metric from the user, enabling implementation flexibility and application portability. Some metrics, for example, may be located in system memory where they are not directly accessible to the application.

MPI/RT probes measure the change in the metric over a portion of the execution of the application. In this way, the application can be partitioned and profiled, for example. Probes are only able to examine local metrics. Probes are not able to access metrics on remote processes.

In addition to the MPI/RT probes, users may define their own probes and notifies MPI/RT via registry, providing additional flexibility.

Both objects defined in this section: MPIRT_PROBE and MPIRT_REGISTRY are not committable.

15.2 MPI/RT Metrics

A quality MPI/RT implementation will accumulate metrics on the behavior of the system. These metrics include the number of buffers sent or received, the number of buffers that failed the QoS, and the total time the buffers took to transmit for each channel. Additional metrics may be tracked, if desired.

A metric is uniquely defined by a {MPIRT_Object, Name} pair. Thus, the number of buffers output over a specific channel can be maintained and retrieved, for example. For system-wide metrics, the MPIRT_OBJECT_NULL is used to specify the metric.

The minimum set of metrics is as follows:

Minimal Set of Metrics			
MPIRT_Object	*Metric Name*	*Description*	*Type*
MPIRT_CHANNEL	"buffer count"	the number of buffers sent or received	int
MPIRT_CHANNEL	"QoS failures"	the number of buffers that failed the QoS	int
MPIRT_CHANNEL	"transmission time"	the aggregate time to send or receive buffers	timespec
MPIRT_BUFITER	"insertion count"	the number of buffers added	int
MPIRT_OBJECT_NULL	"timer"	the timer	timespec

An implementation may choose to implement additional metrics.

Advice to implementors. It is recommended that any implementation-specific counters be implemented as `int`s to simplify application processing of these metrics. (*End of advice to implementors.*)

15.3 MPI/RT Probes

Probes are normally attached to metrics, although a mechanism is defined so that application-specific probes may be created. Probes are used to retrieve the value of the metric and perform some preliminary processing of this value. Because the metric may be located in system memory, the probe hides the actual mechanism for accessing the metric from the user.

Once a probe has been created, *mark* and *accumulate* functions are used to measure the metric over specific areas of the application. *Mark* simply retrieves the current value of the associated metric and saves it in the probe. *Accumulate* retrieves the current value of the metric, subtracts the previously marked value, and adds this result to the accumulated value in the probe. The retrieved value is then saved as the mark value. There is also a function to return the accumulated value of the probe to the application.

In this way, paired mark and accumulate functions can measure the contribution of a selected portion of the application to the overall value of the metric. Also, if mark is never used, the accumulate computation results in simply placing the current value of the metric in the accumulator.

Multiple probes can be attached to the same metric, enabling the measurement of the contribution of a specific portion of an application to the overall metric. For example, two probes may be used to measure the number of buffers sent over a channel: one measures the total number of buffers sent, and the other measures only those buffers sent in a critical portion of the application.

Probes may be put into object containers. Mark and accumulate functions may be performed on the containers; these operations simply iterate over the contents of the container and perform the mark and accumulate functions on each probe in the container.

The following function creates a probe and attaches it to a metric:

MPIRT_PROBE_CREATE (name, mpirt_object, metric_name, probe)

IN	name	name of the probe (MPIRT_STRING_NAME)
IN	mpirt_object	object containing the metric (MPIRT_OBJECT)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
OUT	probe	probe object (ref. to MPIRT_PROBE)

```
int MPIRT_Probe_create (MPIRT_String_name name, MPIRT_Object object,
              MPIRT_String_name metric_name, MPIRT_Probe *probe)
```

```
MPIRT::Probe::Create (const MPIRT::String_Name name, const MPIRT::Object_Type
              &object, const MPIRT::String_Name metric_name)
```

The MPIRT_OBJECT is the object containing the metric, and the metric_name is the name of the metric to manipulate. Some metrics are generic and not specific to an object. For these metrics, an MPIRT_OBJECT_NULL is used.

Because some implementations will have more than the required metrics, it is not an error to specify a metric that doesn't exist on a specific implementation. Application portability is maintained by returning an MPIRT_PROBE_NULL object instead.

Also note that the same metric name can be used for different object types. The *object-type* and name pair uniquely define the semantics of the metric.

The following functions retrieve information from the probe object:

MPIRT_PROBE_GET_OBJECT (probe, object)

IN	probe	probe object (MPIRT_PROBE)
OUT	object	object containing the metric (ref. to MPIRT_OBJECT)

```
int MPIRT_Probe_get_object (MPIRT_Probe probe, MPIRT_Object *object)
```

```
int MPIRT::Probe::Get_Object (MPIRT::Object *&object) const
```

MPIRT_PROBE_GET_METRIC (probe, metric)

IN	probe	probe object (MPIRT_PROBE)
OUT	metric	metric name (ref. to MPIRT_STRING_NAME)

```
int MPIRT_Probe_get_metric (MPIRT_Probe probe, MPIRT_String_name *metric)
```

```
int MPIRT::Probe::Get_Metric (MPIRT::String_Name &metric) const
```

The following function retrieves the dataspec of the accumulated value in the probe. MPI/RT defines several standard dataspecs.

MPIRT_PROBE_RETRIEVE_DATASPEC (probe, type)

IN	probe	probe object (MPIRT_PROBE)
OUT	type	MPI/RT dataspec (ref. to MPIRT_DATASPEC)

```
int MPIRT_Probe_retrieve_dataspec (MPIRT_Probe probe, MPIRT_Dataspec *type)
```

```
int MPIRT::Probe::Retrieve_Dataspec (MPIRT::Dataspec *&type) const
```

The following function returns the number of dataspecs in the accumulated value.

MPIRT_PROBE_RETRIEVE_DATACOUNT (probe, count)

IN	probe	probe object (MPIRT_PROBE)
OUT	count	MPI/RT data count (ref. to integer)

```
int MPIRT_Probe_retrieve_datacount (MPIRT_Probe probe, int *count)
```

```
int MPIRT::Probe::Retrieve_Datacount (int &count) const
```

The following function returns a pointer to the accumulated value:

MPIRT_PROBE_RETRIEVE_VALUE (probe, value)

IN	probe	probe object (MPIRT_PROBE)
OUT	value	accumulated probe value (ref. to MPIRT_ADDRESS)

```
int MPIRT_Probe_retrieve_value (MPIRT_Probe probe, MPIRT_Address *value)
```

```
int MPIRT::Probe::Retrieve_Value (MPIRT::Address &value) const
```

Note that this function returns a reference to the actual accumulator, not a copy. As a result, the accumulator value can be cleared or otherwise modified through this reference.

The following functions set values in the probe object.

MPIRT_PROBE_SET_OBJECT (probe, object)

INOUT	probe	probe object (ref. to MPIRT_PROBE)
IN	object	MPI/RT object (MPIRT_OBJECT)

```
int MPIRT_Probe_set_object (MPIRT_Probe *probe, MPIRT_Object object)
```

```
int MPIRT::Probe::Set_Object (const MPIRT_Object &object)
```

If MPIRT_PROBE_SET_OBJECT attempts to change the kind of object being probed, MPIRT_ERR_TYPE_CHANGE is returned. For example, if a probe is attached to an object of type MPIRT_CHANNEL, it cannot be set to an object of type MPIRT_BUFITER.

MPIRT_PROBE_SET_METRIC (probe, metric)

INOUT	probe	probe object (ref. to MPIRT_PROBE)
IN	metric	metric name (MPIRT_STRING_NAME)

```
int MPIRT_Probe_set_metric (MPIRT_Probe *probe, MPIRT_String_name metric)
```

```
int MPIRT::Probe::Set_Metric (const MPIRT::String_Name metric)
```

Advice to implementors. When the metric name is set in a system probe, the other parameters, such as the mark and accumulate functions, may have to be reset. (*End of advice to implementors.*)

The next function marks the current value of a metric. It is implemented by simply recording the current value of the metric. The initial mark for all probes is zero. Marking an already marked probe is valid.

It is valid to call MPIRT_PROBE_MARK for the MPIRT_PROBE_NULL object.

MPIRT_PROBE_MARK (probe)

INOUT	probe	probe object (ref. to MPIRT_PROBE)

```
int MPIRT_Probe_mark (MPIRT_Probe *probe)
```

```
int MPIRT::Probe::Mark(void)
```

The next function acquires the current value of the metric, subtracts the mark value saved in the probe, and adds this result to the accumulator probe value. The acquired value is then saved as the new mark. In this way, the contribution of the repetitive execution of a section of code can be accumulated. Notice that if MPIRT_PROBE_ACCUMULATE is called on a probe that was never marked, the value returned is that which has been accumulated since the creation of the metric. Also note that the final result of repetitive calls to accumulate without an intervening mark is equivalent to doing only the final call to accumulate.

It is valid to call MPIRT_PROBE_ACCUMULATE for the MPIRT_PROBE_NULL object.

MPIRT_PROBE_ACCUMULATE (probe)

INOUT	probe	probe object (ref. to MPIRT_PROBE)

```
int MPIRT_Probe_accumulate (MPIRT_Probe *probe)
```

```
int MPIRT::Probe::Accumulate(void)
```

For programs that run for extended periods, it is possible that counters may overflow. In MPIRT_PROBE_ACCUMULATE, if the current value of the metric is less than the mark value, or the accumulated value overflows the available precision, MPIRT_PROBE_ACCUMULATE returns MPIRT_ERR_PROBE_OVERFLOW.

The following example demonstrates how two probes can be attached to the same metric. The total number of buffers sent is measured, as well as the number of important buffers.

```
MPIRT_Probe_create("Total", output_channel, "buffer count", total_buffers);
MPIRT_Probe_create("Important", output_channel, "buffer count", important_buffers);
..
MPIRT_Probe_mark(&total_buffers);
while(busy)
{
   /* do some I/O */
   MPIRT_Probe_mark(&important_buffers);
   /* do some important I/O */
   MPIRT_Probe_accumulate(&important_buffers);
}
MPIRT_Probe_accumulate(&total_buffers);
MPIRT_Probe_get_value(total_buffers, &total);
MPIRT_Probe_get_value(important_buffers, &important);
printf("%d out of %d buffers were important\n", important, total);
```

⊤ (final)

The duplication, free, and equality testing operations and the get and set name parameter operations for event delivery objects are inherited from the general class of MPIRT_OBJECT (see Section 2.2).

⊥ (final)

15.4 MPI/RT User Metrics

The *user metric* provides a mechanism to extend the capability of the probes to application-specific or unanticipated metrics. The user metric is created by the following function:

MPIRT_REGISTRY_ADD_METRIC (object_type, metric_name, dataspec, datacount, mark_fn, accum_fn)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
IN	dataspec	data type of the accumulator (MPIRT_DATASPEC)
IN	datacount	number of dataspecs (integer)
IN	mark_fn	function to read the metric (MPIRT_PROBE_MARK_FN)
IN	accum_fn	function to accumulate the metric (MPIRT_PROBE_ACCUM_FN)

```
int MPIRT_Registry_add_metric (MPIRT_Object_Type object_type, MPIRT_String_name
              metric_name, MPIRT_Dataspec dataspec, int datacount,
              MPIRT_Probe_mark_fn mark_fn, MPIRT_Probe_accum_fn accum_fn)
```

```
int MPIRT::Registry::Add_Metric (const MPIRT::Object_Type &object_type, const
              MPIRT::String_Name metric_name, const MPIRT::Dataspec &dataspec, int
              datacount, MPIRT::Probe::Mark_Fn mark_fn, MPIRT::Probe::Accum_Fn
              accum_fn)
```

The data type is for the convenience of the user and is not checked or used by MPIRT_REGISTRY_ADD_METRIC. This enables a generic reporting routine to be written if desired. The probe name, metric name, and object name can all be retrieved with MPI/RT calls. The data type for the accumulator must be known in order to format the data correctly. This provides enough information to produce a useful report generation routine without having to write code for each probe. If an application-specific dataspec is created, it must have a value different from the MPI/RT dataspecs. ⊤ (final)

The MPIRT_REGISTRY object is a singleton object that is created by the MPIRT_INIT and is destroyed by the MPIRT_FINALIZE. The MPIRT_REGISTRY can not be created, freed or duplicated by a user. The MPIRT_REGISTRY object also does not have a name parameter and *set* and *get* operation for it. It also does not support equality testing and object decoration since the object is a singleton. ⊥ (final)

Type datacount is the number of dataspecs to be allocated and must be a positive integer. It must be specified correctly. It is used to allocate memory for the accumulator and mark values.

The mark functor is called explicitly by MPIRT_PROBE_MARK and likely implicitly by MPIRT_PROBE_ACCUMULATE. Following are language independent specifications of what a user mark functor and accum functor definition should be. They are not functions defined in MPI/RT. The prototypes for these functions are as follows:

MPIRT_PROBE_MARK_FN (probe, datum)

INOUT	probe	probe object (ref. to MPIRT_PROBE)
INOUT	datum	location to save the current metric (MPIRT_ADDRESS)

```
typedef int (*MPIRT_Probe_mark_fn) (MPIRT_Probe *probe, MPIRT_Address *datum);
```

```
typedef int (*MPIRT::Probe::Mark_Fn) (MPIRT::Probe &probe,
              MPIRT::Address &datum);
```

MPIRT_PROBE_ACCUM_FN (probe, datum)

INOUT	probe	probe object (ref. to MPIRT_PROBE)
INOUT	datum	location of the current mark (MPIRT_ADDRESS)

```
typedef int (*MPIRT_Probe_accum_fn) (MPIRT_Probe *probe, MPIRT_Address *datum);
```

```
typedef int (*MPIRT::Probe::Accum_Fn) (const MPIRT::Probe &probe, const
             MPIRT::Address &datum);
```

An MPIRT_PROBE_ACCUM_FN will typically look like the following:

```
user_accum_fn(probe, mark)
{
  metric_dataspec temp;                       /* allocate metric temp */
  MPIRT_PROBE_RETRIEVE_VALUE(probe, &value); /* find accumulated metric */
  user_mark_fn(probe, &temp);                 /* get current value of metric */
  *value += temp - **mark;                    /* accumulate metric */
  **mark = temp;                              /* update mark */
  return (MPIRT_SUCCESS);
}
```

The following functions retrieve information about the metric:

MPIRT_REGISTRY_GET_DATASPEC (object_type, metric_name, dataspec)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
OUT	dataspec	data type (ref. to MPIRT_DATASPEC)

```
int MPIRT_Registry_get_dataspec (MPIRT_Object_type object_type, MPIRT::String_Name
             metric_name, MPIRT_Dataspec *dataspec)
```

```
int MPIRT::Registry::Get_Dataspec (MPIRT::Object_Type object_type, const
             MPIRT::String metric_name, MPIRT::Dataspec *&dataspec)
```

MPIRT_REGISTRY_GET_DATACOUNT (object_type, metric_name, datacount)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
OUT	datacount	count of data type (ref. to integer)

```
int MPIRT_Registry_get_datacount (MPIRT_Object_type object_type, MPIRT_String_name
             metric_name, int *datacount)
```

```
int MPIRT::Registry::Get_Datacount (MPIRT::Object_Type object_type, const
             MPIRT::String_Name metric_name, int &datacount)
```

MPIRT_REGISTRY_GET_MARK (object_type, metric_name, mark)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
OUT	mark_fn	mark function (ref. to MPIRT_PROBE_MARK_FN)

```
int MPIRT_Registry_get_mark (MPIRT_Object_type object_type, MPIRT_String_name
              metric_name, MPIRT_Probe_mark_fn *mark)
```

```
int MPIRT::Registry::Get_Mark (MPIRT::Object_Type object_type, const
              MPIRT::String_Name metric_name, MPIRT::Probe::Mark_Fn &mark_fn) const
```

MPIRT_REGISTRY_GET_ACCUM (object_type, metric_name, accum)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
OUT	accum_fn	accumulator function (ref. to MPIRT_PROBE_ACCUM_FN)

```
int MPIRT_Registry_get_accum (MPIRT_Object_type object_type, MPIRT_String_name
              metric_name, MPIRT_Probe_accum_fn *accum)
```

```
int MPIRT::Registry::Get_Accum (MPIRT::Object_Type object_type, const
              MPIRT::String_Name metric_name, MPIRT::Probe::Accum_Fn &accum) const
```

The following functions set values into the metric:

MPIRT_REGISTRY_SET_DATASPEC (object_type, metric_name, dataspec)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
IN	dataspec	data type (MPIRT_DATASPEC)

```
int MPIRT_Registry_set_dataspec (MPIRT_Object_type object_type, MPIRT_String_name
              metric_name, MPIRT_Dataspec dataspec)
```

```
int MPIRT::Registry::Set_Dataspec (MPIRT::Object_Type object_type, const
              MPIRT::String_Name metric_name, const MPIRT::Dataspec &dataspec)
```

MPIRT_REGISTRY_SET_DATACOUNT (object_type, metric_name, datacount)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
IN	datacount	count of data type (integer)

```
int MPIRT_Registry_set_datacount (MPIRT_Object_type object_type, MPIRT_String_name
              metric_name, int datacount)
```

```
int MPIRT::Registry::Set_Datacount (MPIRT::Object_Type object_type, const
             MPIRT::String_Name metric_name, int datacount)
```

MPIRT_REGISTRY_SET_MARK (object_type, metric_name, mark)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
IN	mark_fn	mark function (MPIRT_PROBE_MARK_FN)

```
int MPIRT_Registry_set_mark (MPIRT_Object_type object_type, MPIRT_String_name
             metric_name, MPIRT_Probe_mark_fn *mark)
```

```
int MPIRT::Registry::Set_Mark (MPIRT::Object_Type object_type, const
             MPIRT::String_Name metric_name, MPIRT::Probe::Mark_Fn mark)
```

MPIRT_REGISTRY_SET_ACCUM (object_type, metric_name, accum)

IN	object_type	object type (MPIRT_OBJECT_TYPE)
IN	metric_name	name of the metric (MPIRT_STRING_NAME)
IN	accum_fn	accumulator function (MPIRT_PROBE_ACCUM_FN)

```
int MPIRT_Registry_set_accum (MPIRT_Object_type object_type, MPIRT_String_name
             metric_name, MPIRT_Probe_accum_fn *accum_fn)
```

```
int MPIRT::Registry::Set_Accum (MPIRT::Object_Type object_type, const
             MPIRT::String_Name metric_name, MPIRT::Probe::Accum_Fn accum_fn)
```

The MPIRT_REGISTRY supports an operation for removing a metric from a registry.

Rationale. The MPIRT_REGISTRY object allows users to add and remove metrics thus preventing a memory leak. (*End of rationale.*)

MPIRT_REGISTRY_REMOVE_METRIC (metric_name)

IN	metric_name	name of the metric (MPIRT_STRING_NAME)

```
int MPIRT_Registry_remove_metric (MPIRT_String_name metric_name )
```

```
int MPIRT::Registry::Remove_Metric (const MPIRT::String_Name metric_name)
```

15.5 Summary

This section summarizes enumerations and type definitions that are introduced in this chapter.

Minimal Set of Metrics			
MPIRT_Object	*Metric Name*	*Description*	*Type*
MPIRT_CHANNEL	"buffer count"	the number of buffers sent or received	int
MPIRT_CHANNEL	"QOS failures"	the number of buffers that failed the QoS	int
MPIRT_CHANNEL	"transmission time"	the aggregate time to send or receive buffers	timespec
MPIRT_BUFITER	"insertion count"	the number of buffers added	int
MPIRT_OBJECT_NULL	"timer"	the timer	timespec

Miscellaneous Definitions		
Definition	*What is it*	*How defined?*
MPIRT_PROBE	probe object	probe object
MPIRT_PROBE_NULL	NULL probe object	NULL probe object
MPIRT_OBJECT_TYPE_PROBE	a probe's MPIRT_OBJECT_TYPE	enumeration
MPIRT_PROBE_MARK_FN	a type definition for a user metric mark function	typedef to describe a user function
MPIRT_PROBE_ACCUM_FN	a type definition for a user metric accum function	typedef to describe a user function

Miscellaneous Bindings		
Item	*C Bindings*	*C++ Bindings*
MPIRT_PROBE	MPIRT_Probe	MPIRT::Probe
MPIRT_PROBE_NULL	MPIRT_PROBE_NULL	MPIRT::PROBE_NULL
MPIRT_OBJECT_TYPE_PROBE	MPIRT_OBJECT_TYPE_PROBE	MPIRT::OBJECT_TYPE_PROBE
MPIRT_PROBE_MARK_FN	MPIRT_Probe_mark_fn	MPIRT::Probe::Mark_Fn
MPIRT_PROBE_ACCUM_FN	MPIRT_Probe_accum_fn	MPIRT::Probe::Accum_Fn
MPIRT_ERR_PROBE_OVERFLOW	MPIRT_ERR_PROBE_OVERFLOW	MPIRT::ERR_PROBE_OVERFLOW

Probe Operations		
Language Independent Specification	C	C++
MPIRT_PROBE_CREATE	MPIRT_Probe_create	MPIRT::Probe
MPIRT_PROBE_GET_OBJECT	MPIRT_Probe_get_object	MPIRT::Probe::Get_Object
MPIRT_PROBE_GET_METRIC	MPIRT_Probe_get_metric	MPIRT::Probe::Get_Metric
MPIRT_PROBE_SET_OBJECT	MPIRT_Probe_set_object	MPIRT::Probe::Set_Object
MPIRT_PROBE_SET_METRIC	MPIRT_Probe_set_metric	MPIRT::Probe::Set_Metric
MPIRT_PROBE_RETRIEVE_DATASPEC	MPIRT_Probe_retrieve_dataspec	MPIRT::Probe::Retrieve_Dataspec
MPIRT_PROBE_RETRIEVE_DATACOUNT	MPIRT_Probe_retrieve_datacount	MPIRT::Probe::Retrieve_Datacount
MPIRT_PROBE_RETRIEVE_VALUE	MPIRT_Probe_retrieve_value	MPIRT::Probe::Retrieve_Value
MPIRT_PROBE_MARK	MPIRT_Probe_mark	MPIRT::Probe::Mark
MPIRT_PROBE_ACCUMULATE	MPIRT_Probe_accumulate	MPIRT::Probe::Accumulate
MPIRT_PROBE_GET_NAME	MPIRT_Probe_get_name	MPIRT::Probe::Get_Name
MPIRT_PROBE_SET_NAME	MPIRT_Probe_set_name	MPIRT::Probe::Set_Name
MPIRT_PROBE_DUP	MPIRT_Probe_dup	MPIRT::Probe MPIRT::Probe::Dup MPIRT::Probe::Operator=
MPIRT_PROBE_FREE	MPIRT_Probe_free	MPIRT::∼Probe MPIRT::Probe::Free
MPIRT_PROBE_IS_EQUAL	MPIRT_Probe_is_equal	MPIRT::Probe::Is_equal
MPIRT_PROBE_ATTACH_ATTR	MPIRT_Probe_attach_attr	MPIRT::Probe::Attach_Attr
MPIRT_PROBE_RETRIEVE_ATTR	MPIRT_Probe_retrieve_attr	MPIRT::Probe::Retrieve_Attr
MPIRT_PROBE_DELETE_ATTR	MPIRT_Probe_delete_attr	MPIRT::Probe::Delete_Attr

Registry Operations		
Language Independent Specification	C	C++
MPIRT_REGISTRY_ADD_METRIC	MPIRT_Registry_add_metric	MPIRT::Registry::Add_Metric
MPIRT_REGISTRY_REMOVE_METRIC	MPIRT_Registry_remove_metric	MPIRT::Registry::Remove_Metric
MPIRT_REGISTRY_GET_DATASPEC	MPIRT_Registry_get_dataspec	MPIRT::Registry::Get_Dataspec
MPIRT_REGISTRY_GET_DATACOUNT	MPIRT_Registry_get_datacount	MPIRT::Registry::Get_Datacount
MPIRT_REGISTRY_GET_MARK	MPIRT_Registry_get_mark	MPIRT::Registry::Get_Mark
MPIRT_REGISTRY_GET_ACCUM	MPIRT_Registry_get_accum	MPIRT::Registry::Get_Accum
MPIRT_REGISTRY_SET_DATASPEC	MPIRT_Registry_set_dataspec	MPIRT::Registry::Set_Dataspec
MPIRT_REGISTRY_SET_DATACOUNT	MPIRT_Registry_set_datacount	MPIRT::Registry::Set_Datacount
MPIRT_REGISTRY_SET_MARK	MPIRT_Registry_set_mark	MPIRT::Registry::Set_Mark
MPIRT_REGISTRY_SET_ACCUM	MPIRT_Registry_set_accum	MPIRT::Registry::Set_Accum

Part V

Appendix

Annex A

Return Codes

A.1 Return Codes

This section lists the codes that are returned by the various MPI/RT functions.

MPIRT_CONTINUE - next synchronous handler should be called ⊤ (final2)

MPIRT_ERR_ACTIVE_CHANNEL - calling MPIRT_CHANNEL_ACTIVATE, MPIRT_CHANNEL_START, or MPIRT_CHANNEL_WAIT on a channel that is in MPIRT_CHANNEL_STARTED or MPIRT_CHANNEL_ACTIVATED state

MPIRT_ERR_BUFITER_OVERFLOW - attempting to add a buffer to a full buffer iterator

MPIRT_ERR_BUFITER_UNDERFLOW - attempting to remove a buffer from an empty buffer iterator

MPIRT_ERR_CHANNEL_NOT_COMMITTED - calling MPIRT_CHANNEL_WAIT on an uncommitted channel

MPIRT_ERR_COMMITTED_OBJECT - attempting to modify or free a committed object ⊤ (final2)

⊥ (final2)

MPIRT_ERR_CSET_DUPLICATE_MEMBER - attempting to insert a duplicate member in a set container

MPIRT_ERR_CSET_MEMBER_NOT_FOUND - member not found in a set container

MPIRT_ERR_DATASPEC_MISMATCH - the dataspec of a buffer from the allowed or initial buffer list does not match buffer iterator's dataspec

MPIRT_ERR_ILLEGAL_BUFFER - attempt to insert an invalid buffer into buffer iterator at run-time

MPIRT_ERR_ILLEGAL_BUFITER - one or more requirements for the input and output buffer iterators of a channel failed

MPIRT_ERR_INIT_FAILURE - function MPIRT_INIT failed

MPIRT_ERR_INITIALIZATION - calling an MPI/RT function without having called MPIRT_INIT first

MPIRT_ERR_INSUF_MEMORY - insufficient memory

MPIRT_ERR_INSUF_NETWORK - insufficient network resources to satisfy the commit request

MPIRT_ERR_INV_BUFFER - invalid buffer parameter

MPIRT_ERR_INV_BUFITER - invalid bufiter parameter

MPIRT_ERR_INV_CHANNEL - invalid channel parameter

MPIRT_ERR_INV_CONTAINER - invalid container parameter

MPIRT_ERR_INV_CSET - invalid set container parameter

MPIRT_ERR_INV_CVECTOR - invalid vector container parameter

MPIRT_ERR_INV_GROUP - invalid group parameter

MPIRT_ERR_INV_HANDLER - invalid handler parameter

MPIRT_ERR_INV_KEYVAL - invalid key value

MPIRT_ERR_INV_METRIC - invalid metric parameter

MPIRT_ERR_INV_NAME - invalid object name

MPIRT_ERR_INV_OBJECT - invalid object in parameter

MPIRT_ERR_INV_PARAM - invalid parameter

MPIRT_ERR_INV_POLICY - invalid policy parameter

MPIRT_ERR_INV_PROBE - invalid probe parameter

MPIRT_ERR_INV_QOS - invalid quality-of-service parameter

MPIRT_ERR_INV_RANGE - out-of-range vector index

MPIRT_ERR_NO_RECEPTOR_EXISTS - no explicit receptor exists for event name on this process

MPIRT_ERR_NOT_A_BUFFER - an object in the allowed or initial buffer list was not a buffer

MPIRT_ERR_NOT_MAPPED - the number of buffers at the endpoints do not match as required by mapping policy

⊤ (final2)

MPIRT_ERR_OBJECT_NOT_COMMITTED - attempting an operation that requires a committable object that is committed on uncommitted object

⊥ (final2)

MPIRT_ERR_PROBE_OVERFLOW - overflow of probe counter value

MPIRT_ERR_RECEPTOR_ALREADY_EXISTS - receptor already exists for event name on this process

MPIRT_ERR_SCHEDULE - insufficient computing resources to satisfy the commit request

MPIRT_ERR_TIMEOUT - an MPI/RT function timed out

MPIRT_ERR_TYPE_CHANGE - cannot change the type of object probe is attached to

MPIRT_FAILURE - this MPI/RT function could not successfully perform its duties

MPIRT_SUCCESS - this MPI/RT function was successful

Bibliography

[1] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling and Applications.* Kluwer Academic Publishers, 1997.

[2] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[4] D. Ferrari. A new admission control method for real-time communication in an Internetwork. In D. Son, editor, *Advances in Real-Time Systems.* Prentice Hall, 1995. Chapter 5.

[5] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[6] Rainer Händel and Manfred N. Huber. *Integrated Broadband Networks: An Introduction to ATM-Based Networks.* Addison-Wesley, 1991.

[7] S. P. Harbison and G. L. Steelea. *C A Reference Manual.* Prentice Hall, fourth edition, 1995.

[8] IEEE. *The IEEE Standard Dictionary of Electrical and Electronic Terms.* IEEE, sixth edition, 1997.

[9] ISO/IEC 9945-1. *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1996-07-12. ANSI/IEEE Std 1003.1.

[10] F. Jananian. Run-time monitoring of real-time systems. In D. Son, editor, *Advances in Real-Time Systems.* Prentice Hall, 1995. Chapter 18.

[11] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997.

[12] Hermann Kopetz and Paulo Verissimo. Realtime and dependability concepts. In Sape Mullender, editor, *Distributed Systems, 2nd Edition*, pages 411–446. Addisson-Wesley, 1993. Chapter 16.

[13] A. Mehra, A. Indiresan, and K. G. Shin. Resource management for real-time communication: Making theory meet practice. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications*, pages 130–138, 1996.

[14] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

[15] Bjarne Stroustrup. *The C++ Programmin Language*. Addison-Wesley, third edition, 1997.

[16] A. M. van Tilborg and G. M. Koob. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Kluwer Academic Publishers, 1991.

MPI/RT Errors and Return Codes

This is an index of codes that are returned by the various MPI/RT functions.

MPI/RT Function Index

MPI/RT Entity Index

This is an index of objects, their states and conditions, constants, dataspecs and bindings for all of them.

Index