SANDIA REPORT

SAND2006-0420
Unlimited Release
Printed April 2006

Supersedes SAND99-2959
Dated December 1999

The Portals 3.3 Message Passing Interface Document Revision 2.1

Rolf Riesen, Ron Brightwell, and Kevin Pedretti, Sandia National Laboratories
Arthur B. Maccabe, University of New Mexico,
Trammell Hudson, Rotomotion

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2006-0420
Unlimited Release
Printed April 2006

Supersedes SAND99-2959
dated December 1999

The Portals 3.3 Message Passing Interface
Document Revision 2.1

Rolf Riesen
Ron Brightwell
Kevin Pedretti
Scalable Computing Systems Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110
rolf@cs.sandia.gov
bright@cs.sandia.gov
ktpedre@sandia.gov

Arthur B. Maccabe
Computer Science Department
University of New Mexico
Albuquerque, NM 87131-1386
maccabe@cs.unm.edu

Trammell Hudson
c/o OS Research
1527 16th NW #5
Washington, DC 20036
hudson@osresearch.net

Abstract

This report presents a specification for the portals 3.3 message passing interface. Portals 3.3 are intended to allow scalable, high-performance network communication between nodes of a parallel computing system. Specifically, it is designed to support a parallel computing platform composed of clusters of commodity workstations connected by a commodity system area network fabric. In addition, Portals 3.3 are well suited to massively parallel processing and embedded systems. Portals 3.3 represent an adaption of the data movement layer developed for massively parallel processing platforms, such as the 4500-node Intel TeraFLOPS machine. Version 3.0 of Portals runs on the Cplant cluster at Sandia National Laboratories, and version 3.3 is running on Cray's Red Storm system.

Acknowledgments

Over the years, many people have helped shape, design, and write portals code. We wish to thank: Eric Barton, Peter Braam, Lee Ann Fisk, David Greenberg, Eric Hoffman, Gabi Istrail, Jeanette Johnston, Chu Jong, Clint Kaul, Mike Levenhagen, Kevin McCurley, Jim Otto, David Robboy, Mark Sears, Lance Shuler, Jim Schutt, Mack Stallcup, Todd Underwood, David van Dresser, Dena Vigil, Lee Ward, and Stephen Wheat.

People who were influential in managing the project were: Bill Camp, Ed Barsis, Art Hale, and Neil Pundit

While we have tried to be comprehensive in our listing of the people involved, it is very likely that we have missed at least one important contributor. The omission is a reflection of our poor memories and not a reflection of the importance of their contributions. We apologize to the unnamed contributor(s).

Contents

List of Figures 9

List of Tables 10

List of Implementation Notes 11

Preface 12

Nomenclature 13

1 Introduction 15

1.1 Overview . . . 15
1.2 Purpose . . . 16
1.3 Background . . . 16
1.4 Scalability . . . 17
1.5 Communication Model . . . 17
1.6 Zero Copy, OS Bypass, and Application Bypass . . . 18
1.7 Faults . . . 18

2 An Overview of the Portals API 21

2.1 Data Movement . . . 21
2.2 Portals Addressing . . . 21
2.3 Access Control . . . 24
2.4 Multi-Threaded Applications . . . 26

3 The Portals API 27

3.1 Naming Conventions and Typeface Usage . . . 27
3.2 Base Types . . . 27
3.2.1 Sizes . . . 28
3.2.2 Handles . . . 28
3.2.3 Indexes . . . 28
3.2.4 Match Bits . . . 28
3.2.5 Network Interfaces . . . 28
3.2.6 Identifiers . . . 28
3.2.7 Status Registers . . . 29
3.3 Return Codes . . . 29
3.4 Initialization and Cleanup . . . 29
3.4.1 PtlInit . . . 29
3.4.2 PtlFini . . . 30

3.5 Network Interfaces . . . 30
3.5.1 The Network Interface Limits Type . . . 31
3.5.2 PtlNIInit . . . 32
3.5.3 PtlNIFini . . . 33
3.5.4 PtlNIStatus . . . 33
3.5.5 PtlNIDist . . . 34
3.5.6 PtlNIHandle . . . 35
3.6 User Identification . . . 36
3.6.1 PtlGetUid . . . 36
3.7 Process Identification . . . 36
3.7.1 The Process Identification Type . . . 37
3.7.2 PtlGetId . . . 37
3.8 Process Aggregation . . . 37
3.8.1 PtlGetJid . . . 38
3.9 Match List Entries and Match Lists . . . 38
3.9.1 Match Entry Type Definitions . . . 38
3.9.2 PtlMEAttach . . . 39
3.9.3 PtlMEAttachAny . . . 40
3.9.4 PtlMEInsert . . . 41
3.9.5 PtlMEUnlink . . . 42
3.10 Memory Descriptors . . . 42
3.10.1 The Memory Descriptor Type . . . 43
3.10.2 The Memory Descriptor I/O Vector Type . . . 45
3.10.3 PtlMDAttach . . . 45
3.10.4 PtlMDBind . . . 47
3.10.5 PtlMDUnlink . . . 47
3.10.6 PtlMDUpdate . . . 48
3.10.7 Thresholds and Unlinking . . . 49
3.11 Events and Event Queues . . . 50
3.11.1 Kinds of Events . . . 50
3.11.2 Event Occurrence . . . 51
3.11.3 Event Ordering . . . 53
3.11.4 Failure Notification . . . 53
3.11.5 The Event Queue Type . . . 54
3.11.6 The Event Queue Handler Type . . . 55
3.11.7 PtlEQAlloc . . . 55
3.11.8 Event Queue Handler Semantics . . . 56
3.11.9 PtlEQFree . . . 57
3.11.10 PtlEQGet . . . 58

3.11.11 PtlEQWait . . . 58
3.11.12 PtlEQPoll . . . 59
3.11.13 Event Semantics . . . 60
3.12 The Access Control Table . . . 61
3.12.1 PtlACEntry . . . 61
3.13 Data Movement Operations . . . 62
3.13.1 Portals Acknowledgment Type Definition . . . 62
3.13.2 PtlPut . . . 62
3.13.3 PtlPutRegion . . . 64
3.13.4 PtlGet . . . 65
3.13.5 PtlGetRegion . . . 65
3.13.6 PtlGetPut . . . 66
3.14 Operations on Handles . . . 68
3.14.1 PtlHandleIsEqual . . . 68
3.15 Summary . . . 68

4 The Semantics of Message Transmission **75**
4.1 Sending Messages . . . 75
4.2 Receiving Messages . . . 78

References **81**

Appendix

A Frequently Asked Questions **83**

B Portals Design Guidelines **85**
B.1 Mandatory Requirements . . . 85
B.2 The *Will* Requirements . . . 85
B.3 The *Should* Requirements . . . 86

C A README Template **89**

D Implementations **91**
D.1 Reference Implementation . . . 91
D.2 Portals 3.3 on the Cray XT3 Red Storm . . . 91
D.2.1 Generic . . . 92
D.2.2 Accelerated . . . 92

E Summary of Changes **93**
E.1 Changes From Version 3.0 to 3.1 . . . 93
E.1.1 Thread Issues . . . 93
E.1.2 Handling Small, Unexpected Messages . . . 94
E.1.3 Other Changes . . . 95

E.2 Changes From Version 3.1 to 3.2 96
E.3 Changes From Version 3.2 to 3.3 97
E.3.1 API Changes 97
E.3.2 Semantic Clarifications 97
E.3.3 Document Changes 98

Index **100**

List of Figures

2.1 Portals Put (Send) 22
2.2 **Portals Get.** 23
2.3 **Portals Getput (swap).** 24
2.4 Portals Addressing Structures 25
2.5 **Portals Address Translation.** 26

3.1 Portals Operations and Event Types 52

List of Tables

3.1 Object Type Codes 27
3.2 Memory Descriptor Update Operations 49
3.3 Event Type Summary 53
3.4 Portals Data Types 69
3.5 Portals Functions 70
3.6 Portals Return Codes 71
3.7 Portals Constants 72

4.1 Send Request 76
4.2 Acknowledgment 76
4.3 Get Request 77
4.4 Reply 77
4.5 Get/Put Request 78
4.6 Portals Operations and Memory Descriptor Flags 79

List of Implementation Notes

1 No wire protocol . . . 17
2 User memory as scratch space . . . 18
3 Don't alter put or reply buffers . . . 19
4 Protected space . . . 24
5 Write-only event queue . . . 24
6 README and portals3.h . . . 27
7 Network interface encoded in handle . . . 28
8 Maximum length of **PtlGetPut()** operation . . . 31
9 Multiple calls to **PtlNIInit()** . . . 33
10 Measure of **PtlNIDist()** . . . 35
11 Object encoding in handle . . . 35
12 Checking *match_id* . . . 40
13 Pairing of match list entries and memory descriptors . . . 43
14 Checking legality of md . . . 46
15 Unique memory descriptor handles . . . 48
16 Pending operations and buffer modifications . . . 51
17 Pending operations and *acknowledgment* . . . 52
18 Timing of start events . . . 53
19 Completion of portals operations . . . 54
20 Location of event queue . . . 56
21 Fairness of **PtlEQPoll()** . . . 59
22 Macros using **PtlEQPoll()** . . . 60
23 Filling in the **ptl_event_t** structure . . . 61
24 Functions that require communication . . . 62
25 Information on the wire . . . 75
26 Acknowledgment requests . . . 76

Preface

In the early 1990s, when memory-to-memory copying speeds were an order of magnitude faster than the maximum network bandwidth, it did not matter if data had to go through one or two intermediate buffers on its way from the network into user space. This began to change with early supercomputers, such as the nCUBE 2 and the Intel Paragon, when network bandwidth became similar to memory bandwidth. An intermediate memory-to-memory copy now meant that only half the available bandwidth was used.

Early versions of Portals solved this problem in a novel way. Instead of waiting for data to arrive and then copy it into the final destination, Portals, in versions prior to 3.0, allowed a user to describe what should happen to incoming data by using data structures. A few basic data structures were used like Lego^{tm} blocks to create more complex structures. The operating system kernel handling the data transfer read these structures when data began to arrive and determined where to place the incoming data. Users were allowed to create matching criteria and to specify precisely where data would eventually end up. The kernel, in turn, had the ability to DMA data directly into user space, which eliminated buffer space in kernel owned memory and slow memory-to-memory copies. We named that approach Portals version 2.0. It is still in use today on the ASCI Red supercomputer, the first general-purpose machine to break the one-teraflop barrier.

Although very successful on architectures with lightweight kernels, such as ASCI Red, Portals proved difficult to port to Cplant [Brightwell et al. 2000] with its full-featured Linux kernel. Under Linux, memory was no longer physically contiguous in a one-to-one mapping with the kernel. This made it prohibitively expensive for the kernel to traverse data structures in user space. We wanted to keep the basic concept of using data structures to describe what should happen to incoming data. We put a thin application programming interface (API) over our data structures. We got rid of some never-used building blocks, improved some of the others, and Portals 3.0 were born.

We defined the version 3.0 API in Brightwell, Hudson, Riesen, and Maccabe (1999). Since then, Portals have gone through three revisions. In this report we document version 3.3. The differences between those revisions are explored in Riesen, Brightwell, and Maccabe (2005). Appendix E has a detailed list of changes between the versions.

Version 3.3 is used by the Lustre file system and the Red Storm system by Cray Inc. The development and design of Portals is an ongoing activity at Sandia National Laboratories and the University of New Mexico.

Nomenclature

ACK	Acknowledgement.
FM	Illinois Fast Messages.
AM	Active Messages.
API	Application Programming Interface. A definition of the functions and semantics provided by library of functions.
ASCI	Advanced Simulation and Computing Initiative.
ASCI Red	Intel Tflops system installed at Sandia National Laboratories. First general-purpose system to break one teraflop barrier.
CPU	Central Processing Unit.
DMA	Direct Memory Access.
EQ	Event Queue.
FIFO	First In, First Out.
FLOP	Floating Point OPeration. (Also FLOPS or flops: Floating Point OPerations per Second.)
GM	Glenn's Messages; Myricom's Myrinet API.
ID	Identifier
Initiator	A *process* that initiates a message operation.
IOVEC	Input/Output Vector.
MD	Message Descriptor.
ME	Match list Entry.
Message	An application-defined unit of data that is exchanged between *processes*.
Message Operation	Either a *put* operation, which writes data to a *target*, or a *get* operation, which reads data from a *target*, or a *getput* which does both atomically.
MPI	Message Passing Interface.
MPP	Massively Parallel Processor.
NAL	Network Abstraction Layer.
NAND	Bitwise Not AND operation.
Network	A network provides point-to-point communication between *nodes*. Internally, a network may provide multiple routes between endpoints (to improve fault tolerance or to improve performance characteristics); however, multiple paths will not be exposed outside of the network.
NI	Abstract portals Network Interface.
NIC	Network Interface Card.
Node	A node is an endpoint in a *network*. Nodes provide processing capabilities and memory. A node may provide multiple processors (an SMP node) or it may act as a *gateway* between networks.
OS	Operating System.
PM	Message passing layer for SCoreD [Ishikawa et al. 1996].
POSIX	Portable Operating System Interface.
Process	A context of execution. A process defines a virtual memory context. This context is not shared with other processes. Several threads may share the virtual memory context defined by a process.
RDMA	Remote Direct Memory Access.
RMPP	Reliable Message Passing Protocol.
SMP	Shared Memory Processor.
SUNMOS	Sandia national laboratories/University of New Mexico Operating System.

Target	A *process* that is acted upon by a message operation.
TCP/IP	Transmission Control Protocol/Internet Protocol.
Teraflop	10^{12} flops.
Thread	A context of execution that shares a virtual memory context with other threads.
UDP	User Datagram Protocol.
UNIX	A multiuser, multitasking, portable OS.
VIA	Virtual Interface Architecture.

Chapter 1

Introduction

1.1 Overview

This document describes an application programming interface for message passing between nodes in a system area network. The goal of this interface is to improve the scalability and performance of network communication by defining the functions and semantics of message passing required for scaling a parallel computing system to ten thousand nodes. This goal is achieved by providing an interface that will allow a quality implementation to take advantage of the inherently scalable design of Portals[1].

This document is divided into several sections:

Section 1 – Introduction.
This section describes the purpose and scope of the portals API[2].

Section 2 – An Overview of the Portals 3.3 API.
This section gives a brief overview of the portals API. The goal is to introduce the key concepts and terminology used in the description of the API.

Section 3 – The Portals 3.3 API.
This section describes the functions and semantics of the portals API in detail.

Section 4 – The Semantics of Message Transmission.
This section describes the semantics of message transmission. In particular, the information transmitted in each type of message and the processing of incoming messages.

Appendix A – FAQ.
Frequently Asked Questions about Portals.

Appendix B – Portals Design Guidelines.
The guiding principles behind the portals design.

Appendix C – README-template.
A template for a README file to be provided by each implementation. The README describes implementation specific parameters.

Appendix D – Implementations.
A brief description of the portals 3.3 reference implementation and the implementations that run on Cray's XT3 Red Storm machine.

Appendix E – Summary of Changes.
A list of changes between versions 3.0, 3.1, 3.2, and 3.3.

[1] The word Portals is a plural proper noun. We use it when we refer to the definition, design, version, or similar aspects of Portals.

[2] We use the lower case portals when it is used as an adjective; e.g., portals document, a (generic) portals address, or portals operations. We use the singular when we refer to a specific portal or its attributes; e.g., portal index, portal table, or a (specific) poral address.

1.2 Purpose

Existing message passing technologies available for commodity cluster networking hardware do not meet the scalability goals required by the Cplant [Brightwell et al. 2000] project at Sandia National Laboratories. The goal of the Cplant project is to construct a commodity cluster that can scale to the order of ten thousand nodes. This number greatly exceeds the capacity for which existing message passing technologies have been designed and implemented.

In addition to the scalability requirements of the network, these technologies must also be able to support a scalable implementation of the Message Passing Interface (MPI) [Message Passing Interface Forum 1994] standard, which has become the *de facto* standard for parallel scientific computing. While MPI does not impose any scalability limitations, existing message passing technologies do not provide the functionality needed to allow implementations of MPI to meet the scalability requirements of Cplant.

The following are required properties of a network architecture to avoid scalability limitations:

- Connectionless – Many connection-oriented architectures, such as VIA [Compaq, Microsoft, and Intel 1997] and TCP/IP sockets, have limitations on the number of peer connections that can be established. In Cplant, and other large-scale parallel systems, any node must be able to communicate with any other node without costly connection establishment and tear down.
- Network independence – Many communication systems depend on the host processor to perform operations in order for messages in the network to be consumed. Message consumption from the network should not be dependent on host processor activity, such as the operating system scheduler or user-level thread scheduler. Applications must be able to continue computing while data is moved in and out of the application's memory.
- User-level flow control – Many communication systems manage flow control internally to avoid depleting resources, which can significantly impact performance as the number of communicating processes increases. An application should be able to provide final destination buffers into which the network can deposit data directly.
- OS bypass – High performance network communication should not involve memory copies into or out of a kernel-managed protocol stack. Because networks are now faster or as fast as memory buses, data has to flow directly into user space.

The following are properties of a network architecture that avoids scalability limitations for an implementation of MPI:

- Receiver-managed – Sender-managed message passing implementations require a persistent block of memory to be available for every process, requiring memory resources to increase with job size.
- User-level bypass (application bypass) – While OS bypass is necessary for high performance, it alone is not sufficient to support the *progress rule* of MPI asynchronous operations. After an application has posted a receive, data must be delivered and acknowledged without further intervention from the application.
- Unexpected messages – Few communication systems have support for receiving messages for which there is no prior notification. Support for these types of messages is necessary to avoid flow control and protocol overhead.

1.3 Background

Portals were originally designed for and implemented on the nCube 2 machine as part of the SUNMOS (Sandia/UNM OS) [Maccabe et al. 1994] and Puma [Shuler et al. 1995] lightweight kernel development

projects. Portals went through two design phases [Riesen et al. 2005], the latter one is used on the 4500-node Intel TeraFLOPS machine [Sandia National Laboratories 1996]. Portals have been very successful in meeting the needs of such a large machine, not only as a layer for a high-performance MPI implementation [Brightwell and Shuler 1996], but also for implementing the scalable run-time environment and parallel I/O capabilities of the machine.

The second generation portals implementation was designed to take full advantage of the hardware architecture of large MPP machines. However, efforts to implement this same design on commodity cluster technology identified several limitations due to the differences in network hardware, as well as to shortcomings in the design of Portals. Version 3.0 of Portals addresses this problem by adding a thin API over the portals data structures used to instruct the network on where and how to deliver data.

1.4 Scalability

The primary goal in the design of Portals is scalability. Portals are designed specifically for an implementation capable of supporting a parallel job running on tens of thousands of nodes. Performance is critical only in terms of scalability. That is, the level of message passing performance is characterized by how far it allows an application to scale and not by how it performs in micro-benchmarks (e.g., a two-node bandwidth or latency test).

The portals API is designed to allow for scalability, not to guarantee it. Portals cannot overcome the shortcomings of a poorly designed application program. Applications that have inherent scalability limitations, either through design or implementation, will not be transformed by Portals into scalable applications. Scalability must be addressed at all levels. Portals do not inhibit scalability and do not guarantee it either. No portals operation requires global communication or synchronization.

Similarly, a quality implementation is needed for Portals to be scalable. If the implementation or the network protocols and hardware underneath it cannot scale to 10,000 nodes, then neither Portals nor the application can.

To support scalability, the portals interface maintains a minimal amount of state. Portals provide reliable, ordered delivery of messages between pairs of processes. Portals are connectionless: a process is not required to explicitly establish a point-to-point connection with another process in order to communicate. Moreover, all buffers used in the transmission of messages are maintained in user space. The *target* process determines how to respond to incoming messages, and messages for which there are no buffers are discarded.

IMPLEMENTATION NOTE 1:	No wire protocol This document does not specify a wire protocol. Portals require a reliable communication layer. Whether that is achieved through software or hardware is up to the implementation. For example, in Cplant the reliable message passing protocol (RMPP) [Riesen and Maccabe 2002] is used to make message transmission over Myrinet reliable, while on ASCI Red the hardware is reliable enough to make a separate protocol unnecessary.

1.5 Communication Model

Portals combine the characteristics of both one-sided and two-sided communication. They define a "matching put" operation and a "matching get" operation. The destination of a *put* (or send) is not an

explicit address; instead, each message contains a set of match bits that allow the receiver to determine where incoming messages should be placed. This flexibility allows Portals to support both traditional one-sided operations and two-sided send/receive operations.

Portals allow the *target* to determine whether incoming messages are acceptable. A *target* process can choose to accept message operations from any specific process or can choose to ignore message operations from any specific process.

1.6 Zero Copy, OS Bypass, and Application Bypass

In traditional system architectures, network packets arrive at the network interface card (NIC), are passed through one or more protocol layers in the operating system, and are eventually copied into the address space of the application. As network bandwidth began to approach memory copy rates, reduction of memory copies became a critical concern. This concern led to the development of zero-copy message passing protocols in which message copies are eliminated or pipelined to avoid the loss of bandwidth.

A typical zero-copy protocol has the NIC generate an interrupt for the CPU when a message arrives from the network. The interrupt handler then controls the transfer of the incoming message into the address space of the appropriate application. The interrupt latency, the time from the initiation of an interrupt until the interrupt handler is running, is fairly significant. To avoid this cost, some modern NICs have processors that can be programmed to implement part of a message passing protocol. Given a properly designed protocol, it is possible to program the NIC to control the transfer of incoming messages without needing to interrupt the CPU. Because this strategy does not need to involve the OS on every message transfer, it is frequently called "OS bypass." ST [Task Group of Technical Committee T11 1998], VIA [Compaq, Microsoft, and Intel 1997], FM [Lauria et al. 1998], GM [Myricom, Inc. 1997], PM [Ishikawa et al. 1996], and Portals are examples of OS bypass mechanisms.

Many protocols that support OS bypass still require that the application actively participates in the protocol to ensure progress. As an example, the long message protocol of PM requires that the application receive and reply to a request to put or get a long message. This complicates the runtime environment, requiring a thread to process incoming requests, and significantly increases the latency required to initiate a long message protocol. The portals message passing protocol does not require activity on the part of the application to ensure progress. We use the term "application bypass" to refer to this aspect of the portals protocol.

IMPLEMENTATION NOTE 2:	User memory as scratch space The portals API allows for user memory to be altered. That means an implementation can utilize user memory as scratch space and staging buffers. Only after an operation succeeds and the end event has been posted, must the user memory reflect exactly the data that has arrived.

1.7 Faults

Given the number of components that we are dealing with and the fact that we are interested in supporting applications that run for very long times, failures are inevitable. The portals API recognizes that the underlying transport may not be able to successfully complete an operation once it has been initiated. This is reflected in the fact that the portals API reports two types of events: events indicating the initiation of an operation and events indicating the successful completion of an operation. Every initiation event is

eventually followed by a completion event. Completion events carry a flag which indicates whether the operation completed successfully or not.

Between the time an operation is started and the time that the operation completes (successfully or unsuccessfully), any memory associated with the operation should be considered volatile. That is, the memory may be changed in unpredictable ways while the operation is progressing. Once the operation completes, the memory associated with the operation will not be subject to further modification (from this operation). Notice that unsuccessful operations may alter memory in an essentially unpredictable fashion.

IMPLEMENTATION NOTE 3:	Don't alter put or reply buffers A quality implementation will not alter data in a user buffer that is used in a *put* or *reply* operation. This is independent of whether the operation succeeds or fails.

Chapter 2

An Overview of the Portals API

In this chapter, we give a conceptual overview of the portals API. The goal is to provide a context for understanding the detailed description of the API presented in the next section.

2.1 Data Movement

A portal represents an opening in the address space of a process. Other processes can use a portal to read (*get*), write (*put*), or atomically swap the memory associated with the portal. Every data movement operation involves two processes, the *initiator* and the *target*. The *initiator* is the process that initiates the data movement operation. The *target* is the process that responds to the operation by accepting the data for a *put* operation, replying with the data for a get operation, or both for a *getput* operation.

In this discussion, activities attributed to a process may refer to activities that are actually performed by the process or *on behalf of the process*. The inclusiveness of our terminology is important in the context of *application bypass*. In particular, when we note that the *target* sends a reply in the case of a get operation, it is possible that a reply will be generated by another component in the system, bypassing the application.

Figures 2.1, 2.2, and 2.3 present graphical interpretations of the portals data movement operations: *put* (send), *get*, and *getput* (swap). In the case of a *put* operation, the *initiator* sends a put request ① message to the *target*. The *target* translates the portal addressing information in the request using its local portals structures. The data may be part of the same packet as the put request or it may be in separate packet(s) as shown in Figure 2.1. The portals API does not specify a wire protocol (Section 4). When the data ② has been put into the remote memory descriptor (or been discarded), the *target* optionally sends an acknowledgment ③ message.

Figure 2.2 is a representation of a *get* operation. First, the *initiator* sends a request ① to the *target*. As with the *put* operation, the *target* translates the portal addressing information in the request using its local portals structures. Once it has translated the portal addressing information, the *target* sends a *reply* ② that includes the requested data.

We should note that portals address translations are only performed on nodes that respond to operations initiated by other nodes; i.e., a *target*. Acknowledgments for *put* operations and replies to *get* and *getput* operations bypass the portals address translation structures at the *initiator*.

The third operation, *getput* (swap), is depicted in Figure 2.3. The *initiator* sends a request ①, possibly containing the *put* data ②, to the *target*. The *target* traverses the local portals structures based on the information in the request to find the appropriate user buffer. The *target* then sends the *get* data in a *reply* message ③ back to the *initiator* and deposits the *put* data in the user buffer.

2.2 Portals Addressing

One-sided data movement models (e.g., shmem [Cray Research, Inc. 1994], ST [Task Group of Technical Committee T11 1998], and MPI-2 [Message Passing Interface Forum 1997]) typically use a triple to address memory on a remote node. This triple consists of a process identifier, memory buffer identifier, and offset.

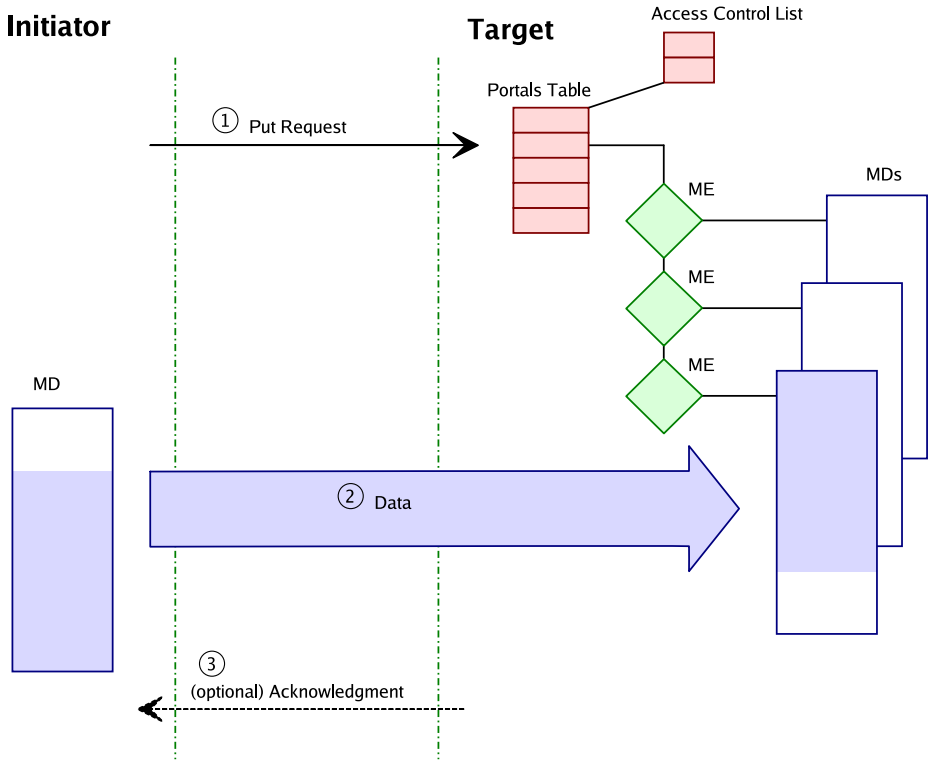


Figure 2.1. Portals Put (Send): Note that the put request ① is part of the header and the data ② is part of the body of a single message. Depending on the network hardware capabilities, the request and data may be sent in a single large packet or several smaller ones.

The process identifier identifies the *target* process, the memory buffer identifier specifies the region of memory to be used for the operation, and the offset specifies an offset within the memory buffer.

In addition to the standard address components (process identifier, memory buffer identifier, and offset), a portals address includes a set of match bits and information identifying the *initiator* (source) of the message. This addressing model is appropriate for supporting one-sided operations, as well as traditional two-sided message passing operations. Specifically, the portals API provides the flexibility needed for an efficient implementation of MPI-1, which defines two-sided operations with one-sided completion semantics.

Figure 2.4 is a graphical representation of the structures used by a *target* in the interpretation of a portals address. The node identifier is used to route the message to the appropriate node and is not reflected in this diagram. The process identifier is used to select the correct *target* process and the portal table it has set up. There is one portal table for each process and each interface initialized by the process; i.e., if a process initializes an interface for a Myrinet and then initializes another interface for an Ethernet, two portal tables will be created within that process, one for each interface. This is not reflected in the diagram.

The portal index is used to select an entry in the portal table. Each entry of the portal table identifies a match list. Each element of the match list specifies two bit patterns: a set of “don’t care” bits and a set of “must match” bits. In addition to the two sets of match bits, each match list entry has at most one memory descriptor. Each memory descriptor identifies a memory region and an optional event queue. The memory region specifies the memory to be used in the operation, and the event queue is used to record information about the operations.

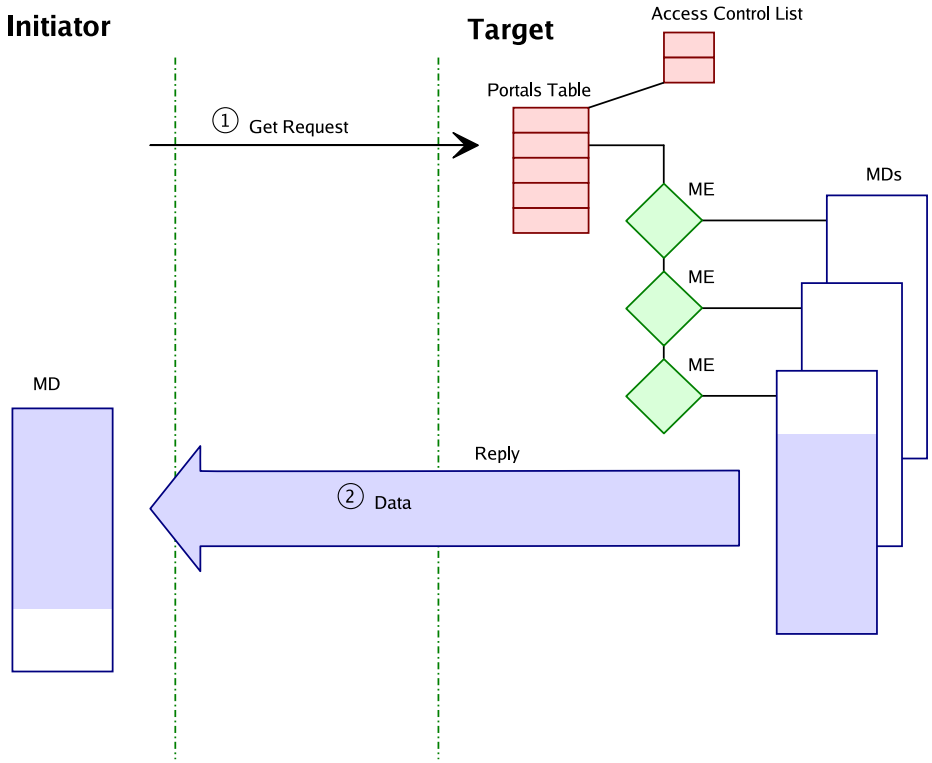


Figure 2.2. Portals Get.

Figure 2.4 illustrates another important concept. The space is divided into protected and application (user) space, while the large data buffers reside in user space. Most of the portals data structures reside in protected space. Often the portals control structures reside inside the operating system kernel or the network interface card. However, they can also reside in a library or another process. See implementation note 20 for possible locations of the event queues.

Figure 2.5 illustrates the steps involved in translating a portals address, starting from the first element in a match list. If the match criteria specified in the match list entry are met and the memory descriptor accepts the operation[1], the operation (*put*, *get*, or *getput*) is performed using the memory region specified in the memory descriptor. Note that matching is done using the match bits, ignore bits, node identifier, and process identifier.

If the memory descriptor specifies that it is to be unlinked when a threshold has been exceeded, the match list entry is removed from the match list, and the resources associated with the memory descriptor and match list entry are reclaimed. If there is an event queue specified in the memory descriptor and the memory descriptor accepts the event, the operation is logged in the event queue. A start event is written before the memory descriptor is altered, and an end event is written when no more actions, as part of the current operation, will be performed on this memory descriptor.

If the match criteria specified in the match list entry are not met, there is no memory descriptor associated with the match list entry, or the memory descriptor associated with the match list entry rejects the operation, the address translation continues with the next match list entry. If the end of the match list has been reached, the address translation is aborted and the incoming requested is discarded.

[1] Memory descriptors can reject operations because a threshold has been exceeded, the memory region does not have sufficient space, or the wrong operation is attempted. See Section 3.10.

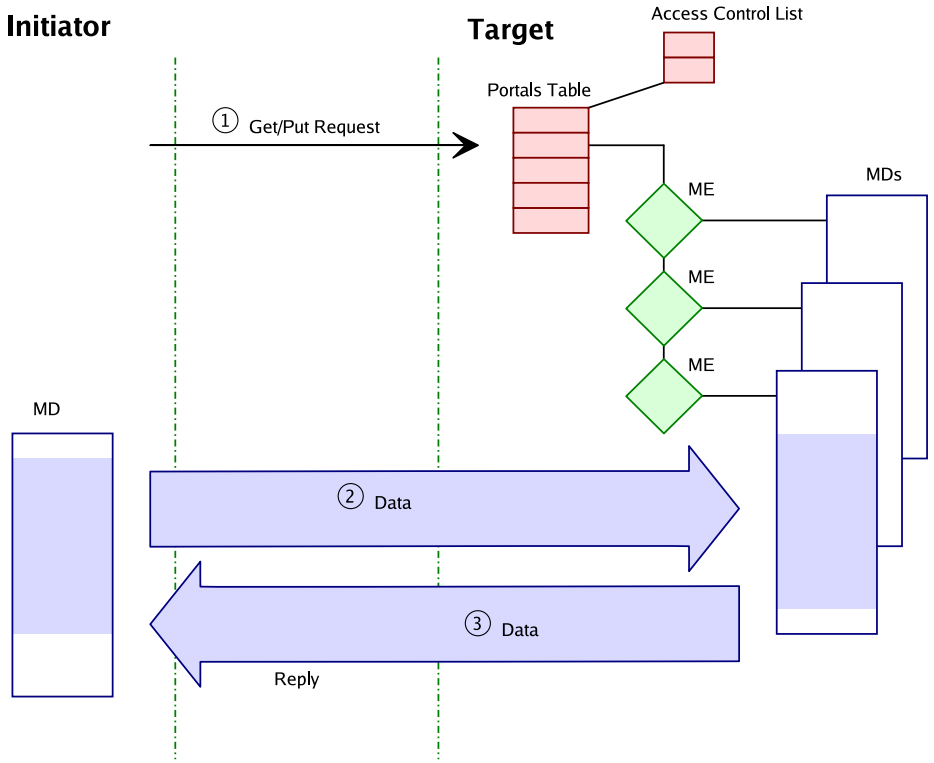


Figure 2.3. Portals Getput (swap).

IMPLEMENTATION NOTE 4:	Protected space Protected space as shown in Figure 2.4 does not mean it has to reside inside the kernel or a different address space. The portals implementation must guarantee that no alterations of portals structures by the user can harm another process or the portals implementation.

IMPLEMENTATION NOTE 5:	Write-only event queue The event queue depicted in Figure 2.4 is a write-only data structure from the point of view of the portals implementation. This avoids reads and locking, which may be expensive from within an NIC.

2.3 Access Control

A process can control access to its portals using an access control list. Each entry in the access control list specifies a process identifier, possibly a job identifier, a user identifier, and a portal table index. The access control list is actually an array of entries. Each incoming request includes an index into the access control list (i.e., a "cookie" or hint). If the identifier of the process issuing the request does not match the identifier

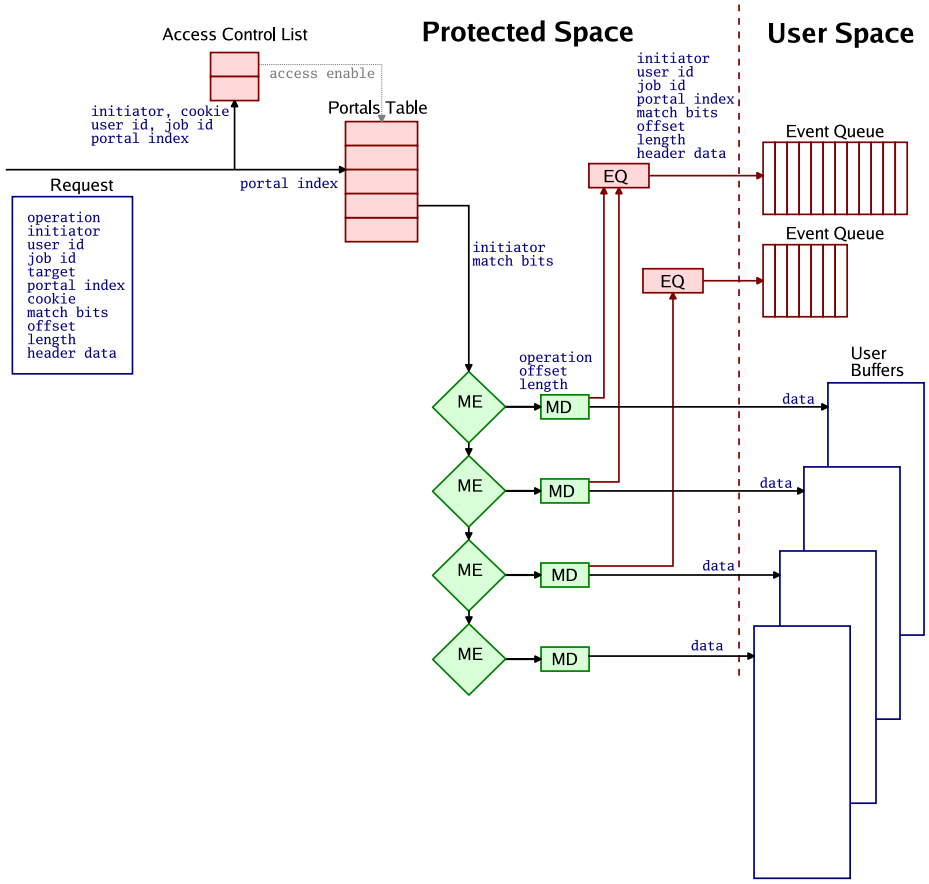


Figure 2.4. Portals Addressing Structures: The example shows two memory descriptors sharing an event queue, one memory descriptor with its own event queue, and a memory descriptor without an event queue. The diagram also shows where incoming header information and data is processed as matching and data deposition take place.

specified in the access control list entry or the portal table index specified in the request does not match the portal table index specified in the access control list entry, the request is rejected. Process identifiers, job identifiers, user identifiers, and portal table indexes may include wildcard values to increase the flexibility of this mechanism.

Two aspects of this design merit further discussion. First, the model assumes that the information identifying the *initiator* in a message header is trustworthy. That information includes the sender's process identifier, node identifier, user identifier, and job identifier. In most contexts, we assume that the entity that constructs the header is trustworthy; however, using cryptographic techniques, we could devise a protocol that would ensure the authenticity of the sender.

Second, because the access check is performed by the receiver, it is possible that a malicious process will generate thousands of messages that will be denied by the receiver. This could saturate the network and/or the receiver, resulting in a *denial of service* attack. Moving the check to the sender using capabilities, would remove the potential for this form of attack. However, the solution introduces the complexities of capability management (exchange of capabilities, revocation, protections, etc). The environments for which Portals were originally designed do not usually have this problem.

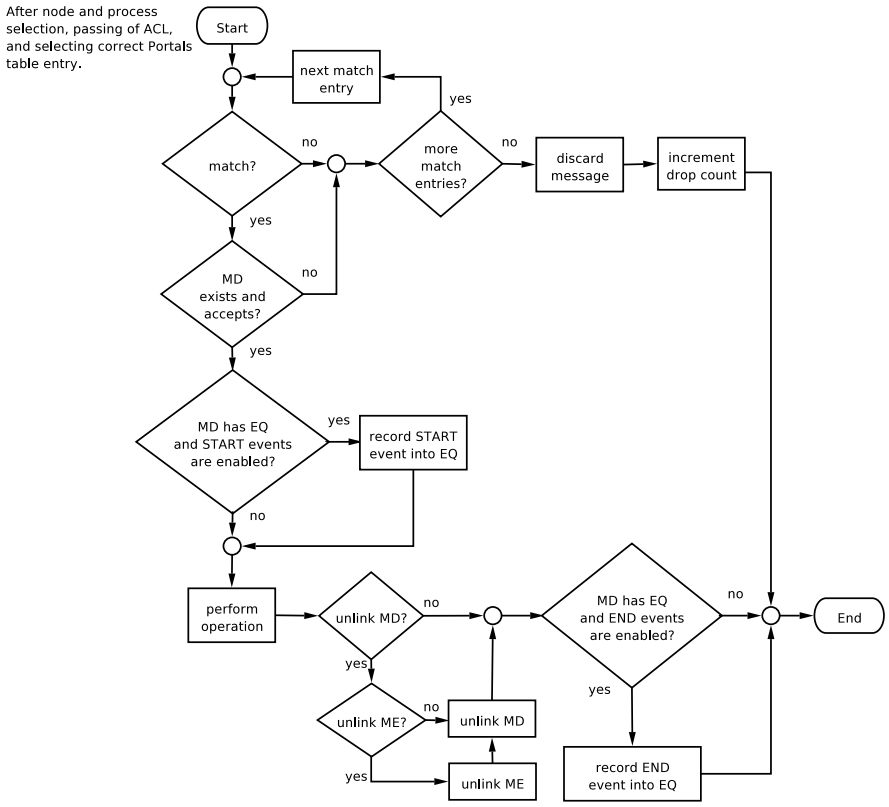


Figure 2.5. Portals Address Translation.

2.4 Multi-Threaded Applications

The portals API supports a generic view of multi-threaded applications. From the perspective of the portals API, an application program is defined by a set of processes. Each process defines a unique address space. The portals API defines access to this address space from other processes (using portals addressing and the data movement operations). A process may have one or more *threads* executing in its address space.

With the exception of **PtlEQWait()** and possibly **PtlEQPoll()**, every function in the portals API is non-blocking and atomic with respect to both other threads and external operations that result from data movement operations. While individual operations are atomic, sequences of these operations may be interleaved between different threads and with external operations. The portals API does not provide any mechanisms to control this interleaving. It is expected that these mechanisms will be provided by the API used to create threads.

Chapter 3

The Portals API

3.1 Naming Conventions and Typeface Usage

The portals API defines four types of entities: functions, types, return codes, and constants. Functions always start with **PtI** and use mixed upper and lower case. When used in the body of this report, function names appear in sans serif bold face, e.g., **PtlInit()**. The functions associated with an object type will have names that start with **PtI**, followed by the two letter object type code shown in column *yy* in Table 3.1. As an example, the function **PtlEQAlloc()** allocates resources for an event queue.

Table 3.1. Object Type Codes.

yy	*xx*	Name	Section
NI	ni	Network Interface	3.5
ME	me	Match list Entry	3.9
MD	md	Memory Descriptor	3.10
EQ	eq	Event Queue	3.11

Type names use lower case with underscores to separate words. Each type name starts with **ptl_** and ends with **_t**. When used in the body of this report, type names appear like this: **ptl_match_bits_t**.

Return codes start with the characters **PTL_** and appear like this: **PTL_OK**.

Names for constants use upper case with underscores to separate words. Each constant name starts with `PTL_`. When used in the body of this report, constant names appear like this: `PTL_ACK_REQ`.

The definition of named constants, function prototypes, and type definitions must be supplied in a file named `portals3.h` that can be included by programs using portals.

IMPLEMENTATION NOTE 6: README and portals3.h

Each implementation must supply an include file named `portals3.h` with the definitions specified in this document. There should also be a README file that explains implementation specific details. For example, it should list the limits (Section 3.5.1) for this implementation and provide a list of status registers that are provided (Section 3.2.7). See Appendix C for a template.

3.2 Base Types

The portals API defines a variety of base types. These types represent a simple renaming of the base types provided by the C programming language. In most cases these new type names have been introduced to improve type safety and to avoid issues arising from differences in representation sizes (e.g., 16-bit or 32-bit integers). Table 3.4 lists all the types defined by Portals.

3.2.1 Sizes

The type **ptl_size_t** is an unsigned 64-bit integral type used for representing sizes.

3.2.2 Handles

Objects maintained by the API are accessed through handles. Handle types have names of the form `ptl_handle_`*xx*`_t`, where *xx* is one of the two letter object type codes shown in Table 3.1, column *xx*. For example, the type **ptl_handle_ni_t** is used for network interface handles. Like all portals types, their names use lower case letters and underscores are used to separate words.

Each type of object is given a unique handle type to enhance type checking. The type **ptl_handle_any_t** can be used when a generic handle is needed. Every handle value can be converted into a value of type **ptl_handle_any_t** without loss of information.

Handles are not simple values. Every portals object is associated with a specific network interface and an identifier for this interface (along with an object identifier) is part of the handle for the object.

IMPLEMENTATION NOTE 7:	Network interface encoded in handle Each handle must encode the network interface it is associated with.

The constant `PTL_EQ_NONE`, of type **ptl_handle_eq_t**, is used to indicate the absence of an event queue. See Sections 3.10.1 and 3.10.6 for uses of this value. The special constant `PTL_INVALID_HANDLE` is used to represent an invalid handle.

3.2.3 Indexes

The types **ptl_pt_index_t** and **ptl_ac_index_t** are integral types used for representing portal table indexes and access control table indexes respectively. See Section 3.5.1 and 3.5.2 for limits on values of these types.

3.2.4 Match Bits

The type **ptl_match_bits_t** is capable of holding unsigned 64-bit integer values.

3.2.5 Network Interfaces

The type **ptl_interface_t** is an integral type used for identifying different network interfaces. Users will need to consult the implementation documentation to determine appropriate values for the interfaces available. The special constant `PTL_IFACE_DEFAULT` identifies the default interface.

3.2.6 Identifiers

The type **ptl_nid_t** is an integral type used for representing node identifiers.**ptl_pid_t** is an integral type for representing process identifiers, **ptl_uid_t** is an integral type for representing user identifiers, and **ptl_jid_t** is an integral type for representing job identifiers.

The special values PTL_PID_ANY matches any process identifier, PTL_NID_ANY matches any node identifier, PTL_UID_ANY matches any user identifier, and PTL_JID_ANY matches any job identifier. See Section 3.12.1 for uses of these values.

3.2.7 Status Registers

Each network interface maintains an array of status registers that can be accessed using the **PtlNISStatus()** function (Section 3.5.4). The type **ptl_sr_index_t** defines the types of indexes that can be used to access the status registers. The only index defined for all implementations is PTL_SR_DROP_COUNT which identifies the status register that counts the dropped requests for the interface. Other indexes (and registers) may be defined by the implementation.

The type **ptl_sr_value_t** defines the types of values held in status registers. This is a signed integer type. The size is implementation dependent but must be at least 32 bits.

3.3 Return Codes

The API specifies return codes that indicate success or failure of a function call. In the case where the failure is due to invalid arguments being passed into the function, the exact behavior of an implementation is undefined. The API suggests error codes that provide more detail about specific invalid parameters, but an implementation is not required to return these specific error codes. For example, an implementation is free to allow the caller to fault when given an invalid address, rather than return **PTL_SEGV**. In addition, an implementation is free to map these return codes to standard return codes where appropriate. For example, a Linux kernel-space implementation could map portals return codes to POSIX-compliant return codes. Table 3.6 lists all return codes used by Portals.

3.4 Initialization and Cleanup

The portals API includes a function, **PtlInit()**, to initialize the library and a function, **PtlFini()**, to clean up after the process is done using the library.

A child process does not inherit any portals resources from its parent. A child process whose parent has initialized portals must shut down and re-initialize portals in order to obtain new, valid portals resources. If a child process fails to shut down and re-initialize portals, behavior is undefined for both the parent and the child.

3.4.1 PtlInit

The **PtlInit()** function initializes the portals library. **PtlInit()** must be called at least once by a process before any thread makes a portals function call but may be safely called more than once.

Function Prototype for PtlInit

```
int PtlInit (int *max_interfaces );
```

Arguments

max_interfaces	**output**	On successful return, this location will hold the maximum number of interfaces that can be initialized.

Return Codes

PTL_OK	Indicates success.
PTL_FAIL	Indicates an error during initialization.
PTL_SEGV	Indicates that *max_interfaces* is not a legal address.

3.4.2 PtlFini

The **PtlFini()** function cleans up after the portals library is no longer needed by a process. After this function is called, calls to any of the functions defined by the portals API or use of the structures set up by the portals API will result in undefined behavior. This function should be called once and only once during termination by a process. Typically, this function will be called in the exit sequence of a process. Individual threads should not call **PtlFini()** when they terminate.

Function Prototype for PtlFini

```
void PtlFini (void);
```

3.5 Network Interfaces

The portals API supports the use of multiple network interfaces. However, each interface is treated as an independent entity. Combining interfaces (e.g., "bonding" to create a higher bandwidth connection) must be implemented by the process or embedded in the underlying network. Interfaces are treated as independent entities to make it easier to cache information on individual network interface cards.

Once initialized, each interface provides a portal table, an access control table, and a collection of status registers. In order to facilitate the development of portable portals applications, a compliant implementation must provide at least 8 portal table entries. See Section 3.9 for a discussion of updating portal table entries using the **PtlMEAttach()** or **PtlMEAttachAny()** functions. See Section 3.12 for a discussion of the initialization and updating of entries in the access control table. See Section 3.5.4 for a discussion of the **PtlNIStatus()** function, which can be used to read the value of a status register.

Every other type of portals object (e.g., memory descriptor, event queue, or match list entry) is associated with a specific network interface. The association to a network interface is established when the object is created and is encoded in the handle for the object.

Each network interface is initialized and shut down independently. The initialization routine, **PtlNIInit()**, returns a handle for an interface object which is used in all subsequent portals operations. The **PtlNIFini()** function is used to shut down an interface and release any resources that are associated with the interface. Network interface handles are associated with processes, not threads. All threads in a process share all of the network interface handles.

The portals API also defines the **PtlNIStatus()** function (Section 3.5.4) to query the status registers for a

network interface, the **PtlNIDist()** function (Section 3.5.5) to determine the "distance" to another process, and the **PtlNIHandle()** function (Section 3.5.6) to determine the network interface with which an object is associated.

3.5.1 The Network Interface Limits Type

The function **PtlNIInit()** accepts a pointer to a list of desired limits and can fill a list with the actual values supported by the network interface. The two lists are of type **ptl_ni_limits_t** and include the following members:

```
typedef struct    {
    int  max_mes;
    int  max_mds;
    int  max_eqs;
    int  max_ac_index;
    int  max_pt_index;
    int  max_md_iovecs;
    int  max_me_list ;
    int  max_getput_md;
} ptl_ni_limits_t   ;
```

Limits

max_mes	Maximum number of match list entries that can be allocated at any one time.
max_mds	Maximum number of memory descriptors that can be allocated at any one time.
max_eqs	Maximum number of event queues that can be allocated at any one time.
max_ac_index	Largest access control table index for this interface, valid indexes range from zero to *max_ac_index*, inclusive.
max_pt_index	Largest portal table index for this interface, valid indexes range from 8 (page 30) to *max_pt_index*, inclusive.
max_md_iovecs	Maximum number of io vectors for a single memory descriptor for this interface.
max_me_list	Maximum number of match entries that can be attached to any portal table index.
max_getput_md	Maximum length, in bytes, of the local and remote memory descriptors used in the atomic swap **PtlGetPut()** operation. Minimum is 8.

IMPLEMENTATION NOTE 8: Maximum length of **PtlGetPut()** operation

An implementation has to allow at least 8 bytes in *getput* operations. However, it is unlikely that many implementations will support more than 8-byte memory descriptors.

3.5.2 PtlNIInit

The **PtlNIInit()** function initializes the portals API for a network interface (NI). A process using portals must call this function at least once before any other functions that apply to that interface. For subsequent calls to **PtlNIInit()** from within the same process (either by different threads or the same thread), the desired limits will be ignored and the call will return the existing network interface handle and actual limits.

Function Prototype for PtlNIInit

```
int PtlNIInit (ptl_interface_t      iface ,
               ptl_pid_t            pid,
               ptl_ni_limits_t      *desired ,
               ptl_ni_limits_t      *actual ,
               ptl_handle_ni_t      *ni_handle );
```

Arguments

iface	**input**	Identifies the network interface to be initialized. (See Section 3.2.5 for a discussion of values used to identify network interfaces.)
pid	**input**	Identifies the desired process identifier (for well known process identifiers). The value `PTL_PID_ANY` may be used to let the portals library select a process identifier.
desired	**input**	If not NULL, points to a structure that holds the desired limits.
actual	**output**	If not NULL, on successful return, the location pointed to by actual will hold the actual limits.
ni_handle	**output**	On successful return, this location will hold a handle for the interface.

Discussion: *The use of desired is implementation dependent. In particular, an implementation may choose to ignore this argument*

Discussion: *Each interface has its own sets of limits. In implementations that support multiple interfaces, the limits passed to and returned by **PtlNIInit()** apply only to the interface specified in iface.*

The desired limits are used to offer a hint to an implementation as to the amount of resources needed, and the implementation returns the actual limits available for use. In the case where an implementation does not have any pre-defined limits, it is free to return the largest possible value permitted by the corresponding type (e.g., INT_MAX). A quality implementation will enforce the limits that are returned and take the appropriate action when limits are exceeded, such as using the **PTL_NO_SPACE** return code. The caller is permitted to use maximum values for the desired fields to indicate that the limit should be determined by the implementation.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_IFACE_INVALID	Indicates that *iface* is not a valid network interface.

PTL_NO_SPACE	Indicates that there is insufficient memory to initialize the interface.
PTL_PID_INVALID	Indicates that *pid* is not a valid process identifier.
PTL_SEGV	Indicates that *actual* or *ni_handle* is not a legal address.

IMPLEMENTATION NOTE 9:

Multiple calls to **PtlNIInit()**

If **PtlNIInit()** gets called more than once *per interface*, then the implementation should fill in *actual* and *ni_handle*. It should ignore *pid*. **PtlGetId()** (Section 3.7) can be used to retrieve the *pid*.

3.5.3 PtlNIFini

The **PtlNIFini()** function is used to release the resources allocated for a network interface. Once the **PtlNIFini()** operation has been started, the results of pending API operations (e.g., operations initiated by another thread) for this interface are undefined. Similarly, the effects of incoming operations (*put*, *get*, *getput*) or return values (*acknowledgment* and *reply*) for this interface are undefined.

Function Prototype for PtlNIFini

```
int PtlNIFini ( ptl_handle_ni_ti    ni_handle );
```

Arguments

ni_handle	**input**	A handle for the interface to shut down.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.

3.5.4 PtlNIStatus

The **PtlNIStatus()** function returns the value of a status register for the specified interface. (See Section 3.2.7 for more information on status register indexes and status register values.)

Function Prototype for PtlNIStatus

```
int PtlNIStatus ( ptl_handle_ni_t    ni_handle ,
                  ptl_sr_index_t     status_register ,
                  ptl_sr_value_t     *status );
```

Arguments

ni_handle	**input**	A handle for the interface to use.
status_register	**input**	An index for the status register to read.
status	**output**	On successful return, this location will hold the current value of the status register.

Discussion: *The only status register that must be defined is a drop count register (`PTL_SR_DROP_COUNT`). Implementations may define additional status registers. Identifiers for the indexes associated with these registers should start with the prefix `PTL_SR_`.*

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_SR_INDEX_INVALID	Indicates that *status_register* is not a valid status register.
PTL_SEGV	Indicates that *status* is not a legal address.

3.5.5 PtlNIDist

The **PtlNIDist()** function returns the distance to another process using the specified interface. Distances are only defined relative to an interface. Distance comparisons between different interfaces on the same process may be meaningless.

Function Prototype for PtlNIDist

```
int  PtlNIDist ( ptl_handle_ni_t      ni_handle ,
                 ptl_process_id_t     process ,
                 unsigned long        *distance );
```

Arguments

ni_handle	**input**	A handle for the interface to use.
process	**input**	An identifier for the process whose distance is being requested.
distance	**output**	On successful return, this location will hold the distance to the remote process.

Discussion: *This function should return a static measure of distance. Examples include minimum latency, the inverse of available bandwidth, or the number of switches between the two endpoints.*

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_PROCESS_INVALID	Indicates that *process* is not a valid process identifier.
PTL_SEGV	Indicates that *distance* is not a legal address.

IMPLEMENTATION NOTE 10: Measure of **PtlNIDist()**

An implementation should state in its documentation what measure, if any, is returned by **PtlNIDist()**. (Appendix C.)

3.5.6 PtlNIHandle

The **PtlNIHandle()** function returns a handle for the network interface with which the object identified by *handle* is associated. If the object identified by *handle* is a network interface, this function returns the same value it is passed.

Function Prototype for PtlNIHandle

```
int PtlNIHandle(ptl_handle_any_t    handle,
                ptl_handle_ni_t     *ni_handle);
```

Arguments

handle	**input**	A handle for the object.
ni_handle	**output**	On successful return, this location will hold a handle for the network interface associated with *handle*.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_HANDLE_INVALID	Indicates that *handle* is not a valid handle.
PTL_SEGV	Indicates that *ni_handle* is not a legal address.

IMPLEMENTATION NOTE 11: Object encoding in handle

Every handle should encode the network interface and the object identifier relative to this handle. Both are presumably encoded using integer values.

3.6 User Identification

Every process runs on behalf of a user. User identifiers travel in the trusted portion of the header of a portals message. They can be used at the *target* to limit access via an access control list (Section 3.12).

3.6.1 PtlGetUid

The **PtlGetUid()** function is used to retrieve the user identifier of a process.

Function Prototype for PtlGetUid

```
int  PtlGetUid(ptl_handle_ni_t    ni_handle ,
               ptl_uid_t          *uid);
```

Arguments

ni_handle	**input**	A network interface handle.
uid	**output**	On successful return, this location will hold the user identifier for the calling process.

Discussion: *Note that user identifiers are dependent on the network interface(s). In particular, if a node has multiple interfaces, a process may have multiple user identifiers.*

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_SEGV	Indicates that *uid* is not a legal address.

3.7 Process Identification

Processes that use the portals API can be identified using a node identifier and process identifier. Every node accessible through a network interface has a unique node identifier and every process running on a node has a unique process identifier. As such, any process in the computing system can be uniquely identified by its node identifier and process identifier.

The portals API defines a type, **ptl_process_id_t** for representing process identifiers, and a function, **PtlGetId()**, which can be used to obtain the identifier of the current process.

Discussion: *The portals API does not include thread identifiers. Messages are delivered to processes (address spaces) not threads (contexts of execution).*

3.7.1 The Process Identification Type

The **ptl_process_id_t** type uses two identifiers to represent a process identifier: a node identifier *nid* and a process identifier *pid*.

```
typedef struct    {
    ptl_nid_t   nid;
    ptl_pid_t   pid;
} ptl_process_id_t  ;
```

3.7.2 PtlGetId

Function Prototype for PtlGetId

```
int  PtlGetId (ptl_handle_ni_t        ni_handle ,
               ptl_process_id_t       *id);
```

Arguments

ni_handle	**input**	A network interface handle.
id	**output**	On successful return, this location will hold the identifier for the calling process.

Discussion: *Note that process identifiers are dependent on the network interface(s). In particular, if a node has multiple interfaces, it may have multiple process identifiers.*

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_SEGV	Indicates that *id* is not a legal address.

3.8 Process Aggregation

It is useful in the context of a parallel machine to represent all of the processes in a parallel job through an aggregate identifier. The portals API provides a mechanism for supporting such job identifiers for these systems. However, job identifiers need not be supported by all systems. In order to be fully supported, job identifiers must be included as a trusted part of a message header, as described in Section 2.3.

The job identifier is an opaque identifier shared between all of the distributed processes of an application running on a parallel machine. All application processes and job-specific support programs, such as the parallel job launcher, share the same job identifier. This identifier is assigned by the runtime system upon

job launch and is guaranteed to be unique among application jobs across the entire distributed system. An individual serial process may be assigned a job identifier that is not shared with any other processes in the system or the constant PTL_JID_NONE can be returned.

Implementations that do not support job identifiers should return the value PTL_JID_NONE when **PtlGetId()** is called.

3.8.1 PtlGetJid

Function Prototype for PtlGetJid

```
int PtlGetJid (ptl_handle_ni_t ni_handle,
               ptl_jid_t       *jid );
```

Arguments

ni_handle	**input**	A network interface handle.
jid	**output**	On successful return, this location will hold the job identifier for the calling process.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates the *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_SEGV	Indicates that *jid* is not a legal address.

3.9 Match List Entries and Match Lists

A match list is a chain of match list entries. Each match list entry includes a pointer to a memory descriptor and a set of match criteria. The match criteria can be used to reject incoming requests based on process identifier or the match bits provided in the request. A match list is created using the **PtlMEAttach()** or **PtlMEAttachAny()** functions, which create a match list consisting of a single match list entry, attach the match list to the specified portal index, and return a handle for the match list entry. Match entries can be dynamically inserted and removed from a match list using the **PtlMEInsert()** and **PtlMEUnlink()** functions.

3.9.1 Match Entry Type Definitions

The type **ptl_unlink_t** is used to specify what happens when the memory descriptor that belongs to this match list entry is unlinked. If PTL_UNLINK is specified, then the match list entry will be unlinked (removed from the match list and resources freed) when the memory descriptor is unlinked. For match list entries that should remain in the list even after the memory descriptor is unlinked, the value PTL_RETAIN should be used.

Values of the type **ptl_ins_pos_t** are used to control where a new match list entry is inserted. The value

PTL_INS_BEFORE is used to insert the new entry before the current entry or before the head of the list. The value PTL_INS_AFTER is used to insert the new entry after the current entry or after the last item in the list.

```
typedef enum {PTL_RETAIN, PTL_UNLINK} ptl_unlink_t;
typedef enum {PTL_INS_BEFORE, PTL_INS_AFTER} ptl_ins_pos_t;
```

3.9.2 PtlMEAttach

The **PtlMEAttach()** function creates a match list consisting of a single entry and attaches this list to the portal table for *ni_handle*. This function can be used to create a new list, insert a match entry at the beginning of an existing list, or append a match entry at the end of an existing list.

Function Prototype for PtlMEAttach

```
int PtlMEAttach(ptl_handle_ni_t    ni_handle,
                ptl_pt_index_t     pt_index,
                ptl_process_id_t   match_id,
                ptl_match_bits_t   match_bits,
                ptl_match_bits_t   ignore_bits,
                ptl_unlink_t       unlink_op,
                ptl_ins_pos_t      position,
                ptl_handle_me_t    *me_handle);
```

Arguments

ni_handle	**input**	A handle for the interface to use.
pt_index	**input**	The portal table index where the match list should be attached.
match_id	**input**	Specifies the match criteria for the process identifier of the requester. The constants PTL_PID_ANY and PTL_NID_ANY can be used to wildcard either of the identifiers in the **ptl_process_id_t** structure.
match_bits, *ignore_bits*	**input**	Specify the match criteria to apply to the match bits in the incoming request. The *ignore_bits* are used to mask out insignificant bits in the incoming match bits. The resulting bits are then compared to the match list entry's match bits to determine if the incoming request meets the match criteria.
unlink_op	**input**	Indicates the match list entry should be unlinked when the memory descriptor associated with this match list entry is unlinked. (Note that the check for unlinking a match entry only occurs when the memory descriptor is unlinked.) Valid values are PTL_UNLINK and PTL_RETAIN.
position	**input**	Indicates whether the new match entry should be prepended or appended to the existing match list. If there is no existing list, this argument is ignored and the new match entry becomes the only entry in the list. Allowed constants: PTL_INS_BEFORE, PTL_INS_AFTER.
me_handle	**output**	On successful return, this location will hold a handle for the newly created match list entry.

Discussion: *Incoming match bits undergo a logical NAND operation with the ignore bits. The match bits stored in the match list entry undergo the same operation. The two results are then compared. The following code fragment illustrates this:*

```
( incoming_bits  & ~ignore_bits ) == ( match_bits  & ~ignore_bits )
```

An optimized version of that is shown in the following code fragment:

```
(( incoming_bits  ^ match_bits ) & ~ignore_bits ) == 0
```

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_PT_INDEX_INVALID	Indicates that *pt_index* is not a valid portal table index.
PTL_PROCESS_INVALID	Indicates that *match_id* is not a valid process identifier.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the match list entry.
PTL_ME_LIST_TOO_LONG	Indicates that the resulting match list is too long. The maximum length for a match list is defined by the interface.

IMPLEMENTATION NOTE 12:

Checking *match_id*

Checking whether a *match_id* is a valid process identifier may require global knowledge. However, **PtlMEAttach()** is not meant to cause any communication with other nodes in the system. Therefore, **PTL_PROCESS_INVALID** may not be returned in some cases where it would seem appropriate.

3.9.3 PtlMEAttachAny

The **PtlMEAttachAny()** function creates a match list consisting of a single entry and attaches this list to an unused portal table entry for *iface*.

Function Prototype for PtlMEAttachAny

```
int PtlMEAttachAny(ptl_handle_ni_t  ni_handle ,
                   ptl_pt_index_t   *pt_index ,
                   ptl_process_id_t match_id,
                   ptl_match_bits_t match_bits ,
                   ptl_match_bits_t ignore_bits  ,
                   ptl_unlink_t     unlink_op ,
                   ptl_handle_me_t  *me_handle);
```

Arguments

ni_handle	**input**	A handle for the interface to use.
pt_index	**output**	On successful return, this location will hold the portal index where the match list has been attached.
match_id, *match_bits*, *ignore_bits* , *unlink_op*	**input**	See the discussion for **PtlMEAttach()**.
me_handle	**output**	On successful return, this location will hold a handle for the newly created match list entry.

Return Codes

PTL_OK	Indicates success.
PTL_NI_INVALID	Indicates that *iface* is not a valid network interface handle.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_PROCESS_INVALID	Indicates that *match_id* is not a valid process identifier.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the match list entry.
PTL_PT_FULL	Indicates that there are no free entries in the portal table.

3.9.4 PtlMEInsert

The **PtlMEInsert()** function creates a new match list entry and inserts this entry into the match list containing *base*.

Function Prototype for PtlMEInsert

```
int PtlMEInsert(ptl_handle_me_t      base,
                ptl_process_id_t     match_id,
                ptl_match_bits_t     match_bits ,
                ptl_match_bits_t     ignore_bits  ,
                ptl_unlink_t         unlink_op ,
                ptl_ins_pos_t        position  ,
                ptl_handle_me_t      *me_handle);
```

Arguments

base	**input**	A handle for a match entry. The new match entry will be inserted immediately before or immediately after this match entry.
match_id, *match_bits*, *ignore_bits*, *unlink_op*	**input**	See the discussion for **PtlMEAttach()**
position	**input**	Indicates whether the new match entry should be inserted before or after the *base* entry. Allowed constants: `PTL_INS_BEFORE`, `PTL_INS_AFTER`.
me_handle	**input**	See the discussion for **PtlMEAttach()**.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_PROCESS_INVALID	Indicates that *match_id* is not a valid process identifier.
PTL_ME_INVALID	Indicates that *base* is not a valid match entry handle.
PTL_ME_LIST_TOO_LONG	Indicates that the resulting match list is too long. The maximum length for a match list is defined by the interface.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the match entry.

3.9.5 PtlMEUnlink

The **PtlMEUnlink()** function can be used to unlink a match entry from a match list. This operation also releases any resources associated with the match entry. If a memory descriptor is attached to the match entry, then it will be unlinked as well. It is an error to use the match entry handle after calling **PtlMEUnlink()**.

Function Prototype for PtlMEUnlink

```
int PtlMEUnlink(ptl_handle_me_t me_handle);
```

Arguments

me_handle	**input**	A handle for the match entry to be unlinked.

Discussion: *If the memory descriptor attached to this match entry has pending operations; e.g., an unfinished reply operation, then **PtlMEUnlink()** will return **PTL_ME_IN_USE**, and neither the match entry nor the memory descriptor will be unlinked.*

***PtlMEUnlink()** does not generate a `PTL_EVENT_UNLINK` event.*

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_ME_INVALID	Indicates that *me_handle* is not a valid match entry handle.
PTL_ME_IN_USE	Indicates that the match list entry has pending operations and cannot be unlinked.

3.10 Memory Descriptors

A memory descriptor contains information about a region of a process' memory and optionally points to an event queue where information about the operations performed on the memory descriptor are recorded. The portals API provides two operations to create memory descriptors: **PtlMDAttach()** and **PtlMDBind()**; one

operation to update a memory descriptor: **PtlMDUpdate()**; and one operation to unlink and release the resources associated with a memory descriptor: **PtlMDUnlink()**.

IMPLEMENTATION NOTE 13:	Pairing of match list entries and memory descriptors Because match list entries and memory descriptors almost always come in pairs and transfer of them across a protection boundary is often expensive, some implementations choose to combine the two data structures internally.

3.10.1 The Memory Descriptor Type

The **ptl_md_t** type defines the visible parts of a memory descriptor. Values of this type are used to initialize and update the memory descriptors.

```
typedef struct {
    void          *start ;
    ptl_size_t    length ;
    int           threshold ;
    ptl_size_t    max_size;
    unsigned int  options ;
    void          *user_ptr ;
    ptl_handle_eq_t eq_handle;
} ptl_md_t;
```

Members

start, *length*
: Specify the memory region associated with the memory descriptor. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. The *start* member can be NULL provided that the *length* member is zero. Zero-length buffers (NULL MD) are useful to record events. There are no alignment restrictions on the starting address or the length of the region; although unaligned messages may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

threshold
: Specifies the maximum number of operations that can be performed on the memory descriptor. An operation is any action that could possibly generate an event. (See Section 3.11.1 for the different types of events). In the usual case, the threshold value is decremented for each operation on the memory descriptor. When the threshold value is zero, the memory descriptor is *inactive*, and does not respond to operations. A memory descriptor can have an initial threshold value of zero to allow for manipulation of an inactive memory descriptor by the local process. A threshold value of `PTL_MD_THRESH_INF` indicates that there is no bound on the number of operations that may be applied to a memory descriptor. Note that local operations, e.g., **PtlMDUpdate()**, are not applied to the threshold count. Local operations do generate events, however. (Table 3.3.)

max_size	Specifies the largest incoming request that the memory descriptor should be respond to. When the unused portion of a memory descriptor (length - local offset) falls below this value, the memory descriptor becomes *inactive* and does not respond to further operations. This value is only used if the `PTL_MD_MAX_SIZE` option is specified. It is ignored if `PTL_MD_MANAGE_REMOTE` is set.
options	Specifies the behavior of the memory descriptor. The following options can be selected: enable *put* operations (yes or no), enable *get* operations (yes or no), offset management (local or remote), message truncation (yes or no), acknowledgment (yes or no), use scatter/gather vectors, disable start events, and disable end events. Values for this argument can be constructed using a bitwise OR of the following values:
`PTL_MD_OP_PUT`	Specifies that the memory descriptor will respond to *put* operations. By default, memory descriptors reject *put* operations.
`PTL_MD_OP_GET`	Specifies that the memory descriptor will respond to *get* operations. By default, memory descriptors reject *get* operations.
`PTL_MD_MANAGE_REMOTE`	Specifies that the offset used in accessing the memory region is provided by the incoming request. By default, the offset is maintained locally. When the offset is maintained locally, the offset is incremented by the length of the request so that the next operation (*put* and/or *get*) will access the next part of the memory region. Note that only one offset variable exists per memory descriptor. If both *put* and *get* operations are performed on a memory descriptor, the value of that single variable is updated each time.
`PTL_MD_TRUNCATE`	Specifies that the length provided in the incoming request can be reduced to match the memory available in the region. (The memory available in a memory region is determined by subtracting the offset from the length of the memory region.) By default, if the length in the incoming operation is greater than the amount of memory available, the operation is rejected.
`PTL_MD_ACK_DISABLE`	Specifies that an *acknowledgment* should *not* be sent for incoming *put* operations, even if requested. By default, acknowledgments are sent for *put* operations that request an acknowledgment. Acknowledgments are never sent for *get* operations. The data sent in the *reply* serves as an implicit acknowledgment.
`PTL_MD_IOVEC`	Specifies that the `start` argument is a pointer to an array of type **ptl_md_iovec_t** (Section 3.10.2) and the `length` argument is the length of the array. This allows for a scatter/gather capability for memory descriptors. A scatter/gather memory descriptor behaves exactly as a memory descriptor that describes a single virtually contiguous region of memory. The local offset, truncation semantics, etc., are identical.
`PTL_MD_MAX_SIZE`	Specifies that the *max_size* field in the memory descriptor is to be used. This option is ignored if `PTL_MD_MANAGE_REMOTE` is set.
`PTL_MD_EVENT_START_DISABLE`	Specifies that this memory descriptor should not generate start events.
`PTL_MD_EVENT_END_DISABLE`	Specifies that this memory descriptor should not generate end events.

	Note: It is not considered an error to have a memory descriptor that does not respond to either *put* or *get* operations: Every memory descriptor responds to *reply* operations. Nor is it considered an error to have a memory descriptor that responds to both *put* and *get* operations. In fact, a memory descriptor used in a *getput* operation must be configured to respond to both *put* and *get* operations. If both `PTL_MD_EVENT_START_DISABLE` and `PTL_MD_EVENT_END_DISABLE` are specified, no events will be generated. This includes `PTL_EVENT_UNLINK` but not `PTL_EVENT_ACK`. If start or end events (or both) are enabled, then `PTL_EVENT_UNLINK` events will be generated.
user_ptr	A user-specified value that is associated with the memory descriptor. The value does not need to be a pointer, but must fit in the space used by a pointer. This value (along with other values) is recorded in events associated with operations on this memory descriptor[1].
eq_handle	A handle for the event queue used to log the operations performed on the memory region. If this argument is `PTL_EQ_NONE`, operations performed on this memory descriptor are not logged.

3.10.2 The Memory Descriptor I/O Vector Type

The **ptl_md_iovec_t** type is used to describe scatter/gather buffers of a memory descriptor in conjunction with the `PTL_MD_IOVEC` option. The **ptl_md_iovec_t** is intended to be a type definition of the `struct iovec` type on systems that already support this type.

```
typedef struct {
    void        *iov_base;
    ptl_size_t  iov_len;
} ptl_md_iovec_t;
```

3.10.3 PtlMDAttach

The **PtlMDAttach()** operation is used to create a memory descriptor and attach it to a match list entry. An error code is returned if this match list entry already has an associated memory descriptor.

Function Prototype for PtlMDAttach

```
int PtlMDAttach(ptl_handle_me_t     me_handle,
                ptl_md_t            md,
                ptl_unlink_t        unlink_op,
                ptl_handle_md_t     *md_handle);
```

[1] Tying the memory descriptor to a user-defined value can be useful when multiple memory descriptor share the same event queue or when the memory descriptor needs to be associated with a data structure maintained by the process outside of the portals library. For example, an MPI implementation can set the *user_ptr* argument to the value of an MPI Request. This direct association allows for processing of memory descriptor's by the MPI implementation without a table lookup or a search for the appropriate MPI Request.

Arguments

me_handle	**input**	A handle for the match entry that the memory descriptor will be associated with.
md	**input**	Provides initial values for the user-visible parts of a memory descriptor. Other than its use for initialization, there is no linkage between this structure and the memory descriptor maintained by the API.
unlink_op	**input**	A flag to indicate whether the memory descriptor is unlinked when it becomesinactive, either because the operation threshold drops to zero or because the *max_size* threshold value has been exceeded. (Note that the check for unlinking a memory descriptor only occurs after the completion of a successful operation. If the threshold is set to zero during initialization or using **PtlMDUpdate()**, the memory descriptor is **not** unlinked.) Values of the type **ptl_unlink_t** are used to control whether an item is unlinked from a list. The value `PTL_UNLINK` enables unlinking. The value `PTL_RETAIN` disables unlinking.
md_handle	**output**	On successful return, this location will hold a handle for the newly created memory descriptor. The *md_handle* argument can be NULL, in which case the handle will not be returned.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_ME_IN_USE	Indicates that *me_handle* already has a memory descriptor attached.
PTL_ME_INVALID	Indicates that *me_handle* is not a valid match entry handle.
PTL_MD_ILLEGAL	Indicates that *md* is not a legal memory descriptor. This may happen because the memory region defined in *md* is invalid or because the network interface associated with the *eq_handle* in *md* is not the same as the network interface associated with *me_handle*. See implementation note 14.
PTL_EQ_INVALID	Indicates that the event queue associated with *md* is not valid.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the memory descriptor.
PTL_SEGV	Indicates that *md_handle* is not a legal address.

IMPLEMENTATION NOTE 14: Checking legality of md

PtlMDAttach() and the other functions in this section may not be able to determine whether an *md* is legal or not. Therefore, even if **PTL_MD_ILLEGAL** is not returned, an illegal *md* may cause an application to be terminated or behave in an undefined manner later on.

3.10.4 PtlMDBind

The **PtlMDBind()** operation is used to create a "free floating" memory descriptor; i.e., a memory descriptor that is not associated with a match list entry.

Function Prototype for PtlMDBind

```
int  PtlMDBind(ptl_handle_ni_t      ni_handle ,
               ptl_md_t             md,
               ptl_unlink_t         unlink_op ,
               ptl_handle_md_t      *md_handle);
```

Arguments

ni_handle	**input**	A handle for the network interface with which the memory descriptor will be associated.
md, *unlink_op*	**input**	See the discussion for **PtlMDAttach()**.
md_handle	**output**	On successful return, this location will hold a handle for the newly created memory descriptor. The *md_handle* argument must be a valid address and cannot be NULL.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_MD_ILLEGAL	Indicates that *md* is not a legal memory descriptor. This may happen because the memory region defined in *md* is invalid or because the network interface associated with the *eq_handle* in *md* is not the same as the network interface, *ni_handle*. See implementation note 14.
PTL_EQ_INVALID	Indicates that the event queue associated with *md* is not valid.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the memory descriptor.
PTL_SEGV	Indicates that *md_handle* is not a legal address.

3.10.5 PtlMDUnlink

The **PtlMDUnlink()** function unlinks the memory descriptor from any match list entry it may be linked to and releases the internal resources associated with a memory descriptor. (This function does not free the memory region associated with the memory descriptor; i.e., the memory the user allocated for this memory descriptor.) This function also releases the resources associated with a floating memory descriptor. Only memory descriptors with no pending operations may be unlinked. Explicitly unlinking a memory descriptor via this function call has the same behavior as a memory descriptor that has been automatically unlinked, except that no `PTL_EVENT_UNLINK` event is generated.

IMPLEMENTATION NOTE 15:	Unique memory descriptor handles An implementation will be greatly simplified if the encoding of memory descriptor handles does not get reused. This makes debugging easier, and it avoids race conditions between threads calling **PtlMDUnlink()** and **PtlMDBind()**.

Function Prototype for PtlMDUnlink

```
int PtlMDUnlink(ptl_handle_md_t md_handle);
```

Arguments

md_handle	**input**	A handle for the memory descriptor to be released.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor handle.
PTL_MD_IN_USE	Indicates that *md_handle* has pending operations and cannot be unlinked. See Figure 3.1 for when data structures are considered to be in use.

3.10.6 PtlMDUpdate

The **PtlMDUpdate()** function provides a conditional, atomic update operation for memory descriptors. The memory descriptor identified by *md_handle* is only updated if the event queue identified by *eq_handle* is empty. The intent is to only carry out updates to the memory descriptor when no new messages have arrived since the last time the queue was checked.

If *new_md* is not NULL the memory descriptor identified by *md_handle* will be updated to reflect the values in the structure pointed to by *new_md* if *eq_handle* has the value `PTL_EQ_NONE` or if the event queue identified by *eq_handle* is empty. If *old_md* is not NULL, the current value of the memory descriptor identified by *md_handle* is recorded in the location identified by *old_md*. A successful update operation resets the local offset of the memory descriptor.

Function Prototype for PtlMDUpdate

```
int PtlMDUpdate(ptl_handle_md_t md_handle,
                ptl_md_t        *old_md,
                ptl_md_t        *new_md,
                ptl_handle_eq_t eq_handle);
```

Arguments

md_handle	**input**	A handle for the memory descriptor to update.
old_md	**output**	If *old_md* is not NULL, the current value of the memory descriptor will be stored in the location identified by *old_md*.
new_md	**input**	If *new_md* is not NULL, this argument provides the new values for the memory descriptor if the update is performed.
eq_handle	**input**	A handle for an event queue used to predicate the update. If *eq_handle* is equal to PTL_EQ_NONE, the update is performed unconditionally. Otherwise, the update is performed if and only if the queue pointed to by *eq_handle* is empty. If the update is not performed, the function returns the value **PTL_MD_NO_UPDATE**. (Note that the *eq_handle* argument does not need to be the same as the event queue associated with the memory descriptor as long as it belongs to the same network interface as the memory descriptor.)

The conditional update can be used to ensure that the memory descriptor has not changed between the time it was examined and the time it is updated. In particular, it is needed to support an MPI implementation where the activity of searching an unexpected message queue and posting a receive must be atomic.

Table 3.2. Memory Descriptor Update Operations.

old_md	*new_md*	**Operation**
NULL	NULL	n/a
NULL	*new*	atomic set of memory descriptor
old	NULL	read memory descriptor
old	*new*	read and atomic set of memory descriptor

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_NO_UPDATE	Indicates that the update was not performed because *eq_handle* was not empty.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor handle.
PTL_MD_ILLEGAL	Indicates that the value pointed to by *new_md* is not a legal memory descriptor (e.g., the memory region specified by the memory descriptor may be invalid).
PTL_EQ_INVALID	Indicates that *eq_handle* is not a valid event queue handle.
PTL_SEGV	Indicates that *new_md* or *old_md* is not a legal address.

3.10.7 Thresholds and Unlinking

The value of the threshold of a memory descriptor is checked before each operation. If the threshold is non-zero, it is decremented after the operation is initiated. A threshold that has been decremented to zero may still have operations that are pending. If the memory descriptor is configured to automatically unlink,

the unlink event will not be generated until all pending operations have been completed. Binding a new memory descriptor to a match list entry is only permitted after an earlier memory descriptor has been explicitly unlinked — successful **PtlMDUnlink()** — or after an unlink event has been posted.

3.11 Events and Event Queues

Event queues are used to log operations performed on local memory descriptors. In particular, they signal the start and end of a data transmission into or out of a memory descriptor. They can also be used to hold acknowledgments for completed *put* operations and indicate when a memory descriptor has been unlinked. Multiple memory descriptors can share a single event queue. An event queue may have an optional event handler associated with it. If an event handler exists, it will be run for each event that is deposited into the event queue.

In addition to the **ptl_handle_eq_t** type, the portals API defines two types associated with events: The **ptl_event_kind_t** type defines the kinds of events that can be stored in an event queue. The **ptl_event_t** type defines a structure that holds the information associated with an event.

The portals API provides five functions for dealing with event queues: The **PtlEQAlloc()** function is used to allocate the API resources needed for an event queue, the **PtlEQFree()** function is used to release these resources, the **PtlEQGet()** function can be used to get the next event from an event queue, the **PtlEQWait()** function can be used to block a process (or thread) until an event queue has at least one event, and the **PtlEQPoll()** function can be used to test or wait on multiple event queues.

3.11.1 Kinds of Events

The portals API defines twelve types of events that can be logged in an event queue:

```
typedef enum {
    PTL_EVENT_GET_START, PTL_EVENT_GET_END,
    PTL_EVENT_PUT_START, PTL_EVENT_PUT_END,
    PTL_EVENT_GETPUT_START, PTL_EVENT_GETPUT_END,
    PTL_EVENT_REPLY_START, PTL_EVENT_REPLY_END,
    PTL_EVENT_SEND_START, PTL_EVENT_SEND_END,
    PTL_EVENT_ACK, PTL_EVENT_MD_UNLINK
} ptl_event_kind_t ;
```

Event types

`PTL_EVENT_GET_START`	A remote *get* operation has started on the memory descriptor. The memory region associated with this descriptor should not be altered until `PTL_EVENT_GET_END` event is logged.
`PTL_EVENT_GET_END`	A previously initiated *get* operation completed successfully.
`PTL_EVENT_PUT_START`	A remote *put* operation has started on the memory descriptor. The memory region associated with this descriptor should be considered volatile until the corresponding END event is logged.
`PTL_EVENT_PUT_END`	A previously initiated *put* operation completed successfully. The underlying layers will not alter the memory (on behalf of this operation) once this event has been logged.

PTL_EVENT_GETPUT_START	A remote *getput* operation has started on the memory descriptor. The memory region associated with this descriptor should not be altered until the corresponding END event is logged.
PTL_EVENT_GETPUT_END	A previously initiated *getput* operation completed successfully.
PTL_EVENT_REPLY_START	A *reply* operation has started on the memory descriptor.
PTL_EVENT_REPLY_END	A previously initiated *reply* operation has completed successfully . This event is logged after the data (if any) from the reply has been written into the memory descriptor.
PTL_EVENT_SEND_START	An outgoing *send* operation has started. The memory region associated with this descriptor should not be altered until the corresponding END or event is logged.
PTL_EVENT_SEND_END	A previously initiated *send* operation has completed. This event is logged after the entire buffer has been sent and it is safe for the caller to reuse the buffer.
PTL_EVENT_ACK	An *acknowledgment* was received. This event is logged when the acknowledgment is received
PTL_EVENT_UNLINK	A memory descriptor was unlinked (Section 3.10.7 and 3.10.3).

3.11.2 Event Occurrence

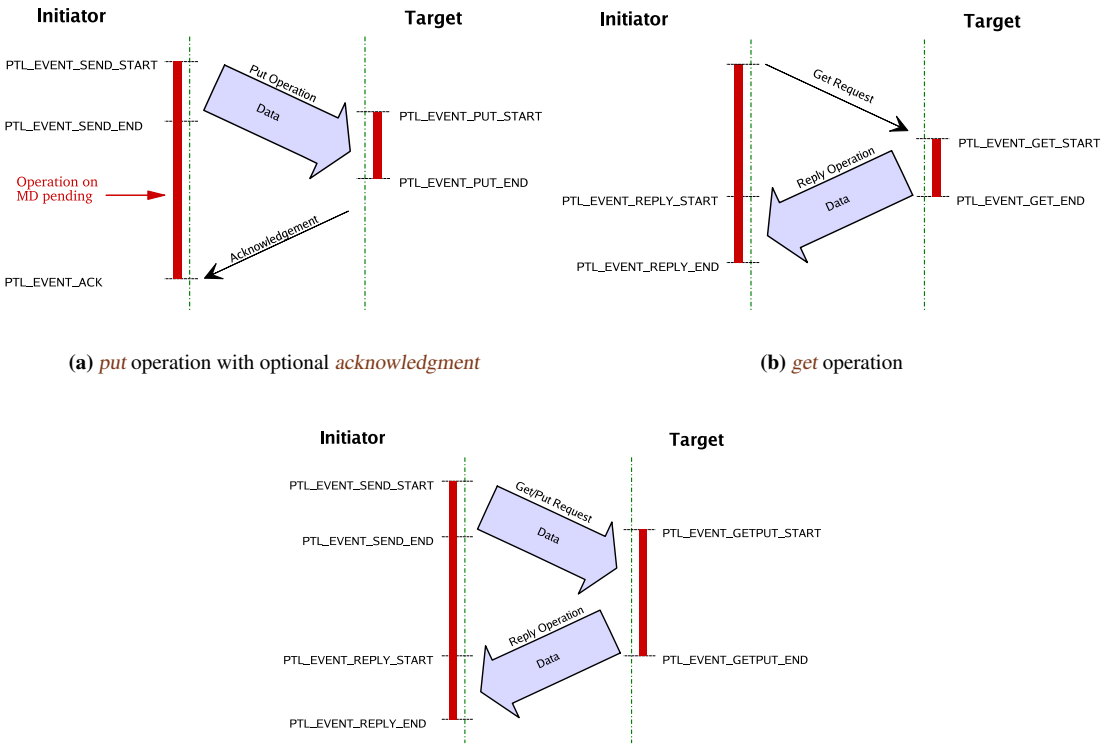
The diagrams in Figure 3.1 show when events occur in relation to portals operations and whether they are recorded on the *initiator* or the *target* side. Note that local and remote events are not synchronized or ordered with respect to each other.

IMPLEMENTATION NOTE 16: Pending operations and buffer modifications

Figure 3.1(a) indicates that the memory descriptor is in use starting at PTL_EVENT_SEND_START until PTL_EVENT_ACK. However, the initiator is free to modify the buffer the memory descriptor describes after the PTL_EVENT_SEND_END event. Also see implementation note 17.

Figure 3.1(a) shows the events that are generated for a *put* operation including the optional *acknowledgment*. The diagram shows which events are generated at the *initiator* and the *target* side of the *put* operation. Figure 3.1(b) shows the corresponding events for a *get* operation, and Figure 3.1(c) shows the events generated for a *getput* operation.

If during any of the operations shown in the diagrams of Figure 3.1, a memory descriptor is unlinked, then a PTL_EVENT_UNLINK event is generated on the *target* or *initiator* where it was unlinked. This is not shown in the diagrams. None of these events are generated, if the memory descriptor has no event queue attached to it (see the description of PTL_EQ_NONE on page 45 of Section 3.10.1). Start or end events can be disabled individually. (See the description of PTL_MD_EVENT_START_DISABLE and PTL_MD_EVENT_END_DISABLE on page 45, also in Section 3.10.1.)



(a) *put* operation with optional *acknowledgment*

(b) *get* operation

(c) *getput* operation

Figure 3.1. Portals Operations and Event Types: The red bars indicate the times a local memory descriptor is considered to be in use by the system; i.e., it has operations pending. Users should not modify memory descriptors during those periods. (Also see implementation notes 16 and 17.)

IMPLEMENTATION NOTE 17:

Pending operations and *acknowledgment*

If a user attempts to unlink a memory descriptor while it has operations pending, the implementation should return **PTL_MD_IN_USE** until the operation has completed or can be aborted cleanly.

After a `PTL_EVENT_SEND_END` a user can attempt to unlink the memory descriptor. If the unlink is successful the implementation should ensure a later acknowledgment is discarded, if it arrives. The same is true for a *reply*, if a successful unlink request occurs before `PTL_EVENT_REPLY_START`. Since users cannot know when events occur, the implementor has a certain amount of freedom honoring unlink requests or returning **PTL_MD_IN_USE**.

Table 3.3 summarizes the portals event types. In the table we use the word *local* to describe the location where the event is delivered; it can be the *initiator* or the *target* of an operation.

Table 3.3. Event Type Summary: A list of event types, where (*initiator* or *target*) they can occur and the meaning of those events.

Event Type	*initiator*	*target*	Meaning
PTL_EVENT_GET_START		•	Data is being "pulled" from a local memory descriptor.
PTL_EVENT_GET_END		•	
PTL_EVENT_PUT_START		•	Data is being "pushed" into a local memory descriptor.
PTL_EVENT_PUT_END		•	
PTL_EVENT_GETPUT_START		•	Data is being "pulled" from and "pushed" into a local memory descriptor.
PTL_EVENT_GETPUT_END		•	
PTL_EVENT_REPLY_START	•		Data is arriving at a local memory descriptor because of a local *get* or *getput* operation.
PTL_EVENT_REPLY_END	•		
PTL_EVENT_SEND_START	•		Data is leaving a local memory descriptor because of a local *put* or *getput* operation.
PTL_EVENT_SEND_END	•		
PTL_EVENT_ACK	•		An acknowledgment has arrived.
PTL_EVENT_UNLINK	•	•	A local memory descriptor has been unlinked.

IMPLEMENTATION NOTE 18: Timing of start events

An implementation can produce a start event as early as during the call that causes it; e.g., **PtlPut()**, but must do so no later than when the data is starting to leave from or arrive in the user buffer.

3.11.3 Event Ordering

As implied by the naming convention, start events must be delivered before end events for a given operation. The portals API also guarantees that when a process initiates two operations on a remote process, the operations will be started on the remote process in the same order that they were initiated on the origin process. As an example, if process A initiates two *put* operations, x and y, on process B, the portals API guarantees that process A will receive the PTL_EVENT_SEND_START events for x and y in the same order that process B receives the PTL_EVENT_PUT_START events for x and y.

Note that memory descriptors that have ignored start or end events using the PTL_MD_EVENT_START_DISABLE or PTL_MD_EVENT_END_DISABLE options are still subject to ordering constraints. Even if the destination memory descriptors for messages x and y have chosen to disable all events, messages x and y must still traverse the portals data structures (e.g., the match list) in the order in which they were initiated.

3.11.4 Failure Notification

Operations may fail to complete successfully; however, unless the node itself fails, every operation that is started will eventually complete. While an operation is in progress, the memory on the *target* associated with the operation should not be viewed (in the case of a *put* or a *reply*) or altered on the *initiator* side (in the case of a *put* or *get*). Operation completion, whether successful or unsuccessful, is final. That is, when an operation completes, the memory associated with the operation will no longer be read or altered by the operation. A network interface can use the integral type **ptl_ni_fail_t** to define specific information regarding the failure of the operation and record this information in the *ni_fail_type* field of an event. The constant PTL_NI_OK should be used in successful start and end events to indicate that there has been no failure.

IMPLEMENTATION NOTE 19:	Completion of portals operations
	Portals guarantees that every operation started will finish with an end event if events are not disabled. While this document cannot enforce or recommend a suitable time, a quality implementation will keep the amount of time between a start and a corresponding end event as short as possible. That includes operations that do not complete successfully. Timeouts of underlying protocols should be chosen accordingly

3.11.5 The Event Queue Type

An event structure includes the following members:

```
typedef struct {
    ptl_event_kind_t        type;
    ptl_process_id_t        initiator ; /* nid, pid */
    ptl_uid_t               uid;
    ptl_jid_t               jid ;
    ptl_pt_index_t          pt_index ;
    ptl_match_bits_t        match_bits ;
    ptl_size_t              rlength ;
    ptl_size_t              mlength;
    ptl_size_t              offset ;
    ptl_handle_md_t         md_handle;
    ptl_md_t                md;
    ptl_hdr_data_t          hdr_data;
    ptl_seq_t               link ;
    ptl_ni_fail_t           ni_fail_type ;
    volatile ptl_seq_t      sequence;
} ptl_event_t ;
```

Members

type	Indicates the type of the event.
initiator	The identifier of the *initiator* (nid, pid).
uid	The user identifier of the *initiator*.
jid	The job identifier of the *initiator*. May be `PTL_JID_NONE` in implementations that do not support job identifiers.
pt_index	The portal table index specified in the request.
match_bits	A copy of the match bits specified in the request. See Section 3.9 for more information on match bits.
rlength	The length (in bytes) specified in the request.
mlength	The length (in bytes) of the data that was manipulated by the operation. For truncated operations, the manipulated length will be the number of bytes specified by the memory descriptor (possibly with an offset) operation. For all other operations, the manipulated length will be the length of the requested operation.

offset	The displacement (in bytes) into the memory region that the operation used. The offset can be determined by the operation (Section 3.13) for a remote managed memory descriptor or by the local memory descriptor (Section 3.10). The offset and the length of the memory descriptor can be used to determine if *max_size* has been exceeded.
md_handle	The handle to the memory descriptor associated with the event. The handle may be invalid if the memory descriptor was unlinked.
md	The state of the memory descriptor immediately after the event has been processed. In particular, the `threshold` field in *md* will reflect the state of the threshold *after* the operation occurred.
hdr_data	64 bits of out-of-band user data (Section 3.13.2).
link	The *link* member is used to link `START` events with the `END` event that signifies completion of the operation. The *link* member will be the same for the two events associated with an operation. The link member is also used to link a `PTL_EVENT_UNLINK` event with the event that caused the memory descriptor to be unlinked.
ni_fail_type	Is used to convey the failure of an operation. Success is indicated by `PTL_NI_OK`. Section 3.11.4.
sequence	The sequence number for this event. Sequence numbers are unique to each event.

Discussion: *The sequence member is the last member and is volatile to support share memory processor (SMP) implementations. When a portals implementation fills in an event structure, the sequence member should be written after all other members have been updated. Moreover, a memory barrier should be inserted between the updating of other members and the updating of the sequence member.*

3.11.6 The Event Queue Handler Type

The **ptl_eq_handler_t** type is used to represent event handler functions. See the discussion in Section 3.11.8 about event queue handler semantics.

```
typedef void (*ptl_eq_handler_t )( ptl_event_t  *event);
```

3.11.7 PtlEQAlloc

The **PtlEQAlloc()** function is used to build an event queue.

Function Prototype for PtlEQAlloc

```
int  PtlEQAlloc(ptl_handle_ni_t      ni_handle ,
                ptl_size_t           count,
                ptl_eq_handler_t     eq_handler,
                ptl_handle_eq_t      *eq_handle);
```

Arguments

ni_handle	**input**	A handle for the interface with which the event queue will be associated.
count	**input**	A hint as to the number of events to be stored in the event queue. An implementation may provide space for more than the requested number of event queue slots.
eq_handler	**input**	A handler function that runs when an event is deposited into the event queue. The constant value `PTL_EQ_HANDLER_NONE` can be used to indicate that no event handler is desired.
eq_handle	**output**	On successful return, this location will hold a handle for the newly created event queue.

Discussion: *An event queue has room for at least count number of events. The event queue is circular and older events will be overwritten by new ones if they are not removed in time by the user — using the functions **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()**. It is up to the user to determine the appropriate size of the event queue to prevent this loss of events.*

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the event queue.
PTL_SEGV	Indicates that *eq_handle* is not a legal address.

IMPLEMENTATION NOTE 20:

Location of event queue

The event queue is designed to reside in user space. High-performance implementations can be designed so they only need to write to the event queue but never have to read from it. This limits the number of protection boundary crossings to update the event queue. However, implementors are free to place the event queue anywhere they like; inside the kernel or the NIC for example.

3.11.8 Event Queue Handler Semantics

The event queue handler, if specified, runs for each event that is deposited into the event queue. The handler is supplied with a pointer to the event that triggered the handler invocation. The handler is invoked at some time between when the event is deposited into the event queue by the underlying communication system and the return of a successful **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()** operation. This implies that if *eq_handler* is not `PTL_EQ_HANDLER_NONE`, **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()** must be called for each event in the queue.

Event handlers may have implementation specific restrictions. In general, handlers must:

- not block;
- not make system calls;
- be reentrant;
- not call **PtlEQWait()**, **PtlEQGet()**, or **PtlEQPoll()**;
- not perform I/O operations; and
- be allowed to call the data movement functions — **PtlPut()**, **PtlPutRegion()**, **PtlGet()**, **PtlGetRegion()**, and **PtlGetPut()**.

Discussion: *An event handler can be called by the implementation when delivering an event or by the portals library when an event is received. In the former case, the implementation must ensure that the address mappings are properly set up for the handler to run. The handler belongs to the address space of the execution thread that called **PtlEQAlloc()**. When run, the handler should not receive any privileges it would not have had if run by the caller of **PtlEQAlloc()**.*

*If handlers are implemented inside the portals library, they must be called before **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()** returns with a status of **PTL_OK** or **PTL_EQ_DROPPED**. Independent of the type of implementation, after a successful handler run, the corresponding event in the event queue is removed.*

*If a handler is specified in **PtlEQAlloc()** (eq_handler $\neq$ PTL_EQ_HANDLER_NONE) and **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()** are not called for every event in the event queue, then behavior is undefined. Behavior is also undefined if a handler does not follow the implementation specific restrictions, for example if a handler blocks.*

3.11.9 PtlEQFree

The **PtlEQFree()** function releases the resources associated with an event queue. It is up to the user to ensure that no memory descriptors are associated with the event queue once it is freed.

Function Prototype for PtlEQFree

```
int PtlEQFree(ptl_handle_eq_t eq_handle);
```

Arguments

eq_handle	**input**	A handle for the event queue to be released.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_INVALID	Indicates that *eq_handle* is not a valid event queue handle.

3.11.10 PtlEQGet

The **PtlEQGet()** function is a nonblocking function that can be used to get the next event in an event queue. If an event handler is associated with the event queue, then the handler will run before this function returns successfully[2] (Section 3.11.8). The event is removed from the queue.

Function Prototype for PtlEQGet

```
int  PtlEQGet(ptl_handle_eq_t      eq_handle,
              ptl_event_t          *event);
```

Arguments

eq_handle	**input**	A handle for the event queue.
event	**output**	On successful return, this location will hold the values associated with the next event in the event queue.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained — using **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()**— from this event queue has been dropped due to limited space in the event queue.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_EMPTY	Indicates that *eq_handle* is empty or another thread is waiting in **PtlEQWait()**.
PTL_EQ_INVALID	Indicates that *eq_handle* is not a valid event queue handle.
PTL_SEGV	Indicates that *event* is not a legal address.

3.11.11 PtlEQWait

The **PtlEQWait()** function can be used to block the calling process or thread until there is an event in an event queue. If an event handler is associated with the event queue, then the handler will run before this function returns successfully (Section 3.11.8). This function returns the next event in the event queue and removes this event from the queue. In the event that multiple threads are waiting on the same event queue, **PtlEQWait()** is guaranteed to wake exactly one thread, but the order in which they are awakened is not specified.

Function Prototype for PtlEQWait

```
int  PtlEQWait(ptl_handle_eq_t     eq_handle,
               ptl_event_t         *event);
```

[2] The handler may have run before the call to **PtlEQGet()**.

Arguments

eq_handle	**input**	A handle for the event queue to wait on. The calling process (thread) will be blocked until the event queue is not empty.
event	**output**	On successful return, this location will hold the values associated with the next event in the event queue.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained — using **PtlEQGet()**, **PtlEQWait()**, or **PtlEQPoll()**— from this event queue has been dropped due to limited space in the event queue.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_INVALID	Indicates that *eq_handle* is not a valid event queue handle.
PTL_SEGV	Indicates that *event* is not a legal address.

3.11.12 PtlEQPoll

The **PtlEQPoll()** function can be used by the calling process to look for an event from a set of event queues. Should an event arrive on any of the queues contained in the array of event queue handles, the event will be returned in *event* and *which* will contain the index of the event queue from which the event was taken.

If an event handler is associated with the event queue, then the handler will run before this function returns successfully (Section 3.11.8). If **PtlEQPoll()** returns success, the corresponding event is consumed. **PtlEQPoll()** provides a timeout to allow applications to poll, block for a fixed period, or block indefinitely. **PtlEQPoll()** is sufficiently general to implement both **PtlEQGet()** and **PtlEQWait()**, but these functions have been retained in the API for backward compatibility.

IMPLEMENTATION NOTE 21: Fairness of **PtlEQPoll()**

PtlEQPoll() should poll the list of queues in a round-robin fashion. This cannot guarantee fairness but meets common expectations.

Function Prototype for PtlEQPoll

```
int PtlEQPoll(ptl_handle_eq_t *eq_handles,
              int             size ,
              ptl_time_t      timeout,
              ptl_event_t     *event,
              int             *which);
```

Arguments

eq_handles	**input**	An array of event queue handles. All the handles must refer to the same interface.
size	**input**	Length of the array.
timeout	**input**	Time in milliseconds to wait for an event to occur on one of the event queue handles. The constant `PTL_TIME_FOREVER` can be used to indicate an infinite timeout.
event	**output**	On successful return (**PTL_OK** or **PTL_EQ_DROPPED**), this location will hold the values associated with the next event in the event queue.
which	**output**	On successful return, this location will contain the index of the event queue from which the event was taken.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained from the event queue indicated by *which* has been dropped due to limited space in the event queue.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_EQ_INVALID	Indicates that one or more of the event queue handles is not valid; e.g., not all handles in *eq_handles* are on the same network interface.
PTL_SEGV	Indicates that *event* or *which* is not a legal address.
PTL_EQ_EMPTY	Indicates that the timeout has been reached and all of the event queues are empty.

IMPLEMENTATION NOTE 22: Macros using **PtlEQPoll()**

Implementations are free to provide macros for **PtlEQGet()** and **PtlEQWait()** that use **PtlEQPoll()** instead of providing these functions.

3.11.13 Event Semantics

The split event sequence is needed to support unreliable networks and/or networks that packetize. The start/end sequence is needed to support networks that packetize where the completion of transfers may not be ordered with initiation of transfers. An implementation is free to implement these event sequences in any way that meets the ordering semantics. For example, an implementation for a network that is reliable and that preserves message ordering (or does not packetize) may generate a start/end event pair at the completion of the transfer. In fact, since the information in the start/end events is identical except for the link field, a correct implementation may generate a single event that the event queue test/wait library function turns into an event pair.

IMPLEMENTATION NOTE 23:	Filling in the **ptl_event_t** structure All of the members of the **ptl_event_t** structure returned from **PtlEQGet()**, **PtlEQWait()**, and **PtlEQPoll()** must be filled in with valid information. An implementation may not leave any field in an event unset.

3.12 The Access Control Table

Processes can use the access control table to control which processes are allowed to perform operations on portal table entries. Each communication interface has a portal table and an access control table. The access control table for the default interface contains an entry at index zero that allows all processes with the same user identifier to communicate. Entries in the access control table can be manipulated using the **PtlACEntry()** function.

3.12.1 PtlACEntry

The **PtlACEntry()** function can be used to update an entry in the access control table for an interface. For those implementations that do not support job identifiers, the *jid* argument is ignored.

Function Prototype for PtlACEntry

```
int PtlACEntry(ptl_handle_ni_t    ni_handle,
               ptl_ac_index_t     ac_index,
               ptl_process_id_t   match_id,
               ptl_uid_t          uid,
               ptl_jid_t          jid,
               ptl_pt_index_t     pt_index);
```

Arguments

ni_handle	**input**	Identifies the interface to use.
ac_index	**input**	The index of the entry in the access control table to update.
match_id	**input**	Identifies the process(es) that are allowed to perform operations. The constants `PTL_PID_ANY` and `PTL_NID_ANY` can be used to wildcard either of the identifiers in the **ptl_process_id_t** structure.
uid	**input**	Identifies the user that is allowed to perform operations. The value `PTL_UID_ANY` can be used to wildcard the user.
jid	**input**	Identifies the collection of processes allowed to perform an operation. The value `PTL_JID_ANY` can be used to wildcard the job identifier.
pt_index	**input**	Identifies the portal index(es) that can be used. The value `PTL_PT_INDEX_ANY` can be used to wildcard the portal index.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_NI_INVALID	Indicates that *ni_handle* is not a valid network interface handle.
PTL_AC_INDEX_INVALID	Indicates that *ac_index* is not a valid access control table index.
PTL_PROCESS_INVALID	Indicates that *match_id* is not a valid process identifier.
PTL_PT_INDEX_INVALID	Indicates that *pt_index* is not a valid portal table index.

3.13 Data Movement Operations

The portals API provides five data movement operations: **PtlPut()**, **PtlPutRegion()**, **PtlGet()**, **PtlGetRegion()**, and **PtlGetPut()**.

IMPLEMENTATION NOTE 24: Functions that require communication

Other than **PtlPut()**, **PtlPutRegion()**, **PtlGet()**, **PtlGetRegion()**, and **PtlGetPut()**, no function in the portals API requires communication with other nodes in the system.

3.13.1 Portals Acknowledgment Type Definition

Values of the type **ptl_ack_req_t** are used to control whether an acknowledgment should be sent when the operation completes (i.e., when the data has been written to a memory descriptor of the *target* process). The value `PTL_ACK_REQ` requests an acknowledgment, the value `PTL_NO_ACK_REQ` requests that no acknowledgment should be generated.

typedef enum {PTL_ACK_REQ, PTL_NO_ACK_REQ} **ptl_ack_req_t**;

3.13.2 PtlPut

The **PtlPut()** function initiates an asynchronous *put* operation. There are several events associated with a *put* operation: initiation of the send on the *initiator* node (`PTL_EVENT_SEND_START`), completion of the send on the *initiator* node (`PTL_EVENT_SEND_END`), and when the send completes successfully, the receipt of an acknowledgment (`PTL_EVENT_ACK`) indicating that the operation was accepted by the *target*. The events `PTL_EVENT_PUT_START` and `PTL_EVENT_PUT_END` are used at the *target* node to indicate begin and end of data delivery. (Figure 3.1.)

These (local) events will be logged in the event queue associated with the memory descriptor (*md_handle*) used in the *put* operation. Using a memory descriptor that does not have an associated event queue results in these events being discarded. In this case, the caller must have another mechanism (e.g., a higher level protocol) for determining when it is safe to modify the memory region associated with the memory descriptor.

Function Prototype for PtlPut

```
int PtlPut (ptl_handle_md_t   md_handle,
            ptl_ack_req_t     ack_req,
            ptl_process_id_t  target_id ,
            ptl_pt_index_t    pt_index ,
            ptl_ac_index_t    ac_index,
            ptl_match_bits_t  match_bits ,
            ptl_size_t        remote_offset ,
            ptl_hdr_data_t    hdr_data);
```

Arguments

md_handle	**input**	A handle for the memory descriptor that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent (PTL_EVENT_SEND_START, PTL_EVENT_SEND_END, PTL_EVENT_ACK).
ack_req	**input**	Controls whether an acknowledgment event is requested. Acknowledgments are only sent when they are requested by the initiating process **and** the memory descriptor has an event queue **and** the target memory descriptor enables them. Allowed constants: PTL_ACK_REQ, PTL_NO_ACK_REQ.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
ac_index	**input**	The index into the access control table of the *target* process.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory descriptor (only used when the *target* memory descriptor has the PTL_MD_MANAGE_REMOTE option set).
hdr_data	**input**	64 bits of user data that can be included in the message header. This data is written to an event queue entry at the *target* if an event queue is present on the matching memory descriptor.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.13.3 PtlPutRegion

The **PtlPutRegion()** function is identical to the **PtlPut()** function except that it allows a region of memory within the memory descriptor to be sent rather than the entire memory descriptor. The local (*initiator*) offset is used to determine the starting address of the memory region and the length specifies the length of the region in bytes. It is an error for the local offset and length parameters to specify memory outside the memory described by the memory descriptor.

Function Prototype for PtlPutRegion

```
int PtlPutRegion(ptl_handle_md_t    md_handle,
                 ptl_size_t         local_offset,
                 ptl_size_t         length,
                 ptl_ack_req_t      ack_req,
                 ptl_process_id_t   target_id,
                 ptl_pt_index_t     pt_index,
                 ptl_ac_index_t     ac_index,
                 ptl_match_bits_t   match_bits,
                 ptl_size_t         remote_offset,
                 ptl_hdr_data_t     hdr_data);
```

Arguments

md_handle	**input**	A handle for the memory descriptor that describes the memory to be sent.
local_offset	**input**	Offset from the start of the memory descriptor.
length	**input**	Length of the memory region to be sent.
ack_req, *target_id*, *pt_index*, *ac_index*	**input**	See the discussion for **PtlPut()**.
match_bits, *remote_offset*, *hdr_data*	**input**	See the discussion for **PtlPut()**.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_MD_ILLEGAL	Indicates that *local_offset* and *length* specify a region outside the bounds of the memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.13.4 PtlGet

The **PtlGet()** function initiates a remote read operation. There are two event pairs associated with a get operation. When the data is sent from the *target* node, a PTL_EVENT_GET_START / PTL_EVENT_GET_END event pair is registered on the *target* node. When the data is returned from the *target* node, a PTL_EVENT_REPLY_START / PTL_EVENT_REPLY_END event pair is registered on the *initiator* node. (Figure 3.1)

Function Prototype for PtlGet

```
int PtlGet( ptl_handle_md_t    md_handle,
            ptl_process_id_t   target_id  ,
            ptl_pt_index_t     pt_index ,
            ptl_ac_index_t     ac_index ,
            ptl_match_bits_t   match_bits ,
            ptl_size_t         remote_offset );
```

Arguments

md_handle	**input**	A handle for the memory descriptor that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
ac_index	**input**	The index into the access control table of the *target* process.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory descriptor (only used when the target memory descriptor has the PTL_MD_MANAGE_REMOTE option set).

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.13.5 PtlGetRegion

The **PtlGetRegion()** function is identical to the **PtlGet()** function except that it allows a region of memory within the memory descriptor to accept a *reply* rather than the entire memory descriptor. The local (*initiator*) offset is used to determine the starting address of the memory region and the length specifies the length of the region in bytes. It is an error for the local offset and length parameters to specify memory outside the memory described by the memory descriptor.

Function Prototype for PtlGetRegion

```
int PtlGetRegion(ptl_handle_md_t    md_handle,
                 ptl_size_t         local_offset ,
                 ptl_size_t         length ,
                 ptl_process_id_t   target_id ,
                 ptl_pt_index_t     pt_index ,
                 ptl_ac_index_t     ac_index ,
                 ptl_match_bits_t   match_bits ,
                 ptl_size_t         remote_offset );
```

Arguments

md_handle	**input**	A handle for the memory descriptor that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.
local_offset	**input**	Offset from the start of the memory descriptor.
length	**input**	Length of the memory region for the *reply*.
target_id, *pt_index*, *ac_index*	**input**	See discussion for **PtlGet()**.
match_bits, *remote_offset*	**input**	See discussion for **PtlGet()**.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *md_handle* is not a valid memory descriptor.
PTL_MD_ILLEGAL	Indicates that *local_offset* and *length* specify a region outside the bounds of the memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.13.6 PtlGetPut

The **PtlGetPut()** function performs an atomic swap of data at the *target* with the data passed in the *put* memory descriptor. The original contents of the memory region at the *target* are returned in a *reply* message and placed into the *get* memory descriptor of the *initiator*. An implementation may restrict the length of the memory descriptors used in **PtlGetPut()** but must support at least 8 bytes (Section 3.5.1). The *target* memory descriptor must be configured to respond to both *get* operations and *put* operations. The *length* field in the *put_md_handle* is used to specify the size of the request.

There are three event pairs associated with a *get* operation. When data is sent from the *initiator* node, a `PTL_EVENT_SEND_START` /`PTL_EVENT_SEND_END` event pair is registered on the *initiator* node. When the data is sent from the *target* node, a `PTL_EVENT_GETPUT_START` /`PTL_EVENT_GETPUT_END` event pair is registered on the *target* node; and when the data is returned from the *target* node, a `PTL_EVENT_REPLY_START` /`PTL_EVENT_REPLY_END` event pair is registered on the *initiator* node. Note that the

target memory descriptor must have both the PTL_MD_OP_PUT and PTL_MD_OP_GET flags set.

Discussion: *Most implementations will need to temporarily store the incoming data while the old data is being sent back to the initiator. Therefore, an implementation can limit the size of getput operations. The minimum size is 8 bytes. The actual value is returned by the* ***PtlNIInit()*** *call in the variable max_getput_md (Section 3.5.2 and 3.5.1).*

Function Prototype for PtlGetPut

```
int PtlGetPut(ptl_handle_md_t      get_md_handle,
              ptl_handle_md_t      put_md_handle,
              ptl_process_id_t     target_id ,
              ptl_pt_index_t       pt_index ,
              ptl_ac_index_t       ac_index ,
              ptl_match_bits_t     match_bits ,
              ptl_size_t           remote_offset ,
              ptl_hdr_data_t       hdr_data);
```

Arguments

get_md_handle	**input**	A handle for the memory descriptor that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.
put_md_handle	**input**	A handle for the memory descriptor that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
ac_index	**input**	The index into the access control table of the *target* process.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory descriptor (only used when the target memory descriptor has the PTL_MD_MANAGE_REMOTE option set).
hdr_data	**input**	64 bits of user data that can be included in the message header. This data is written to an event queue entry at the *target* if an event queue is present on the matching memory descriptor.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the portals API has not been successfully initialized.
PTL_MD_INVALID	Indicates that *put_md_handle* or *get_md_handle* is not a valid memory descriptor.
PTL_PROCESS_INVALID	Indicates that *target_id* is not a valid process identifier.

3.14 Operations on Handles

Handles are opaque data types. The only operation defined on them by the portals API is a comparison function.

3.14.1 PtlHandleIsEqual

The **PtlHandleIsEqual()** function compares two handles to determine if they represent the same object.

Function Prototype for PtlHandleIsEqual

```
int  PtlHandleIsEqual(ptl_handle_any_t  handle1,
                      ptl_handle_any_t  handle2);
```

Arguments

handle1, *handle2*	**input**	A handle for an object. Either of these handles is allowed to be the constant value, `PTL_INVALID_HANDLE`, which represents the value of an invalid handle.

Discussion: ***PtlHandleIsEqual()*** *does not check whether handle1 and handle2 are valid; only whether they are equal.*

Return Codes

PTL_OK	Indicates that the handles are equivalent.
PTL_FAIL	Indicates that the two handles are not equivalent.

3.15 Summary

We conclude this chapter by summarizing the names introduced by the portals API. We start with the data types introduced by the API. This is followed by a summary of the functions defined by the API which is followed by a summary of the function return codes. Finally, we conclude with a summary of the other constant values defined by the API.

Table 3.4 presents a summary of the types defined by the portals API. The first column in this table gives the type name, the second column gives a brief description of the type, the third column identifies the section where the type is defined, and the fourth column lists the functions that have arguments of this type.

Table 3.4. Portals Data Types: Data Types Defined by the Portals API.

Name	Meaning	Sec	Functions
ptl_ac_index_t	access control table indexes	3.2.3	PtlACEntry(), PtlPut(), PtlPutRegion(), PtlGet(), PtlGetRegion(), PtlGetPut()
ptl_ack_req_t	acknowledgment request types	3.13.2	PtlPut(), PtlPutRegion()
ptl_eq_handler_t	event queue handler function	3.11.6	PtlEQAlloc()
ptl_event_kind_t	event kind	3.11.1	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_event_t	event information	3.11.5	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_handle_any_t	any object handles	3.2.2	PtlNIHandle(), PtlHandleIsEqual()
ptl_handle_eq_t	event queue handles	3.2.2	PtlEQAlloc(), PtlEQFree(), PtlEQGet(), PtlEQWait(), PtlEQPoll(), PtlMDUpdate()
ptl_handle_md_t	memory descriptor handles	3.2.2	PtlMDUnlink() , PtlMDUpdate(), PtlMEAttach(), PtlMEAttachAny(), PtlMEInsert(), PtlPut(), PtlPutRegion(), PtlGet(), PtlGetRegion(), PtlGetPut()
ptl_handle_me_t	match entry handles	3.2.2	PtlMEAttach(), PtlMEAttachAny(), PtlMEInsert(), PtlMEUnlink()
ptl_handle_ni_t	network interface handles	3.2.2	PtlNIInit(), PtlNIFini(), PtlNIStatus(), PtlNIDist(), PtlEQAlloc(), PtlACEntry()
ptl_hdr_data_t	user header data	3.13.2	PtlPut(), PtlPutRegion(), PtlGet(), PtlGetRegion(), PtlGetPut()
ptl_ins_pos_t	insert position (before or after)	3.9.2	PtlMEAttach(), PtlMEAttachAny(), PtlMEInsert()
ptl_interface_t	network interface identifiers	3.2.5	PtlNIInit()
ptl_jid_t	job identifier	3.2.6	PtlGetJid(), PtlACEntry()
ptl_match_bits_t	match (and ignore) bits	3.2.4	PtlMEAttach(), PtlMEAttachAny(), PtlMEInsert(), PtlPut(), PtlPutRegion(), PtlGet(), PtlGetRegion(), PtlGetPut()
ptl_md_iovec_t	scatter/gather buffer descriptors	3.10.2	PtlMDAttach(), PtlMDBind(), PtlMDUpdate(), PtlMDUnlink()
ptl_md_t	memory descriptors	3.10.1	PtlMDAttach(), PtlMDBind(), PtlMDUpdate()
ptl_nid_t	node identifiers	3.2.6	PtlGetId(), PtlACEntry()
ptl_ni_fail_t	network interface specific failures	3.11.4	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_ni_limits_t	implementation dependent limits	3.5.1	PtlNIInit()
ptl_pid_t	process identifier	3.2.6	PtlGetId(), PtlACEntry()
ptl_process_id_t	process identifiers	3.7.1	PtlGetId(), PtlNIDist(), PtlMEAttach(), PtlMEAttachAny(), PtlACEntry(), PtlPut(), PtlPutRegion(), PtlGet(), PtlGetRegion(), PtlGetPut()
ptl_pt_index_t	portal table indexes	3.2.3	PtlMEAttach(), PtlMEAttachAny(), PtlPut(), PtlPutRegion(), PtlGet(), PtlGetRegion(), PtlGetPut(), PtlACEntry()
ptl_seq_t	event sequence number	3.11.5	PtlEQGet(), PtlEQWait(), PtlEQPoll()

continued on next page

continued from previous page

Name	Meaning	Sec	Functions
ptl_size_t	sizes	3.2.1	PtlEQAlloc(), PtlPut(), PtlPutRegion(), PtlGet(), PtlGetRegion()
ptl_sr_index_t	status register indexes	3.2.7	PtlNIStatus()
ptl_sr_value_t	status register values	3.2.7	PtlNIStatus()
ptl_time_t	time in milliseconds	3.11.12	PtlEQPoll()
ptl_uid_t	user identifier	3.2.6	PtlGetUid(), PtlACEntry()
ptl_unlink_t	unlink options	3.9.2	PtlMEAttach(), PtlMEAttachAny(), PtlMEInsert(), PtlMDAttach()

Table 3.5 presents a summary of the functions defined by the portals API. The first column in this table gives the name for the function, the second column gives a brief description of the operation implemented by the function, and the third column identifies the section where the function is defined.

Table 3.5. Portals Functions: Functions Defined by the Portals API.

Name	Operation	Definition
PtlACEntry()	update an entry in an access control table	3.12.1
PtlEQAlloc()	create an event queue	3.11.7
PtlEQFree()	release the resources for an event queue	3.11.9
PtlEQGet()	get the next event from an event queue	3.11.10
PtlEQPoll()	poll for a new event on multiple event queues	3.11.12
PtlEQWait()	wait for a new event in an event queue	3.11.11
PtlFini()	shut down the portals API	3.4.2
PtlGet()	perform a *get* operation	3.13.4
PtlGetId()	get the identifier for the current process	3.7.2
PtlGetJid()	get the job identifier for the current process	3.8.1
PtlGetPut()	perform an atomic swap operation	3.13.6
PtlGetRegion()	perform a *get* operation on a memory descriptor region	3.13.5
PtlGetUid()	get the network interface specific user identifier	3.6.1
PtlHandleIsEqual()	compares two handles to determine if they represent the same object	3.14.1
PtlInit()	initialize the portals API	3.4.1
PtlMDAttach()	create a memory descriptor and attach it to a match entry	3.10.3
PtlMDBind()	create a free-floating memory descriptor	3.10.4
PtlMDUnlink()	remove a memory descriptor from a list and release its resources	3.10.5
PtlMDUpdate()	update a memory descriptor	3.10.6
PtlMEAttachAny()	create a match entry and attach it to a free portal table entry	3.9.3
PtlMEAttach()	create a match entry and attach it to a portal table	3.9.2
PtlMEInsert()	create a match entry and insert it in a list	3.9.4
PtlMEUnlink()	remove a match entry from a list and release its resources	3.9.5
PtlNIDist()	get the distance to another process	3.5.5
PtlNIFini()	shut down a network interface	3.5.3
PtlNIHandle()	get the network interface handle for an object	3.5.6
PtlNIInit()	initialize a network interface	3.5.2
PtlNIStatus()	read a network interface status register	3.5.4
PtlPut()	perform a *put* operation	3.13.2
PtlPutRegion()	perform a *put* operation on a memory descriptor region	3.13.3

Table 3.6 summarizes the return codes used by functions defined by the portals API. The first column of this table gives the symbolic name for the constant, the second column gives a brief description of the value, and the third column identifies the functions that can return this value.

Table 3.6. Portals Return Codes: Function Return Codes for the Portals API.

Name	Meaning	Functions
PTL_AC_INDEX_INVALID	invalid access control table index	**PtlACEntry()**
PTL_EQ_DROPPED	at least one event has been dropped	**PtlEQGet()**, **PtlEQWait()**
PTL_EQ_EMPTY	no events available in an event queue	**PtlEQGet()**
PTL_EQ_INVALID	invalid event queue handle	**PtlMDUpdate()**, **PtlEQFree()**, **PtlEQGet()**
PTL_FAIL	error during initialization or cleanup	**PtlInit()**, **PtlFini()**
PTL_HANDLE_INVALID	invalid handle	**PtlNIHandle()**
PTL_IFACE_INVALID	initialization of an invalid interface	**PtlNIInit()**
PTL_MD_ILLEGAL	illegal memory descriptor values	**PtlMDAttach()**, **PtlMDBind()**, **PtlMDUpdate()**
PTL_MD_IN_USE	memory descriptor has pending operations	**PtlMDUnlink()**
PTL_MD_INVALID	invalid memory descriptor handle	**PtlMDUnlink()** , **PtlMDUpdate()**
PTL_MD_NO_UPDATE	no update was performed	**PtlMDUpdate()**
PTL_ME_IN_USE	ME has pending operations	**PtlMEUnlink()**
PTL_ME_INVALID	invalid match entry handle	**PtlMDAttach()**
PTL_ME_LIST_TOO_LONG	match entry list too long	**PtlMEAttach()**, **PtlMEInsert()**
PTL_NI_INVALID	invalid network interface handle	**PtlNIDist()**, **PtlNIFini()**, **PtlMDBind()**, **PtlEQAlloc()**
PTL_NO_INIT	uninitialized API	*all*, except **PtlInit()**
PTL_NO_SPACE	insufficient memory	**PtlNIInit()**, **PtlMDAttach()**, **PtlMDBind()**, **PtlEQAlloc()**, **PtlMEAttach()**, **PtlMEInsert()**
PTL_OK	success	*all*
PTL_PID_INVALID	invalid pid	**PtlNIInit()**
PTL_PROCESS_INVALID	invalid process identifier	**PtlNIInit()**, **PtlNIDist()**, **PtlMEAttach()**, **PtlMEInsert()**, **PtlACEntry()**, **PtlPut()**, **PtlGet()**
PTL_PT_FULL	portal table is full	**PtlMEAttachAny()**
PTL_PT_INDEX_INVALID	invalid portal table index	**PtlMEAttach()**
PTL_SEGV	addressing violation	**PtlNIInit()**, **PtlNIStatus()**, **PtlNIDist()**, **PtlNIHandle()**, **PtlMDBind()**, **PtlMDUpdate()**, **PtlEQAlloc()**, **PtlEQGet()**, **PtlEQWait()**
PTL_SR_INDEX_INVALID	invalid status register index	**PtlNIStatus()**

Table 3.7 summarizes the remaining constant values introduced by the portals API. The first column in this table presents the symbolic name for the constant, the second column gives a brief description of the value, the third column identifies the type for the value, and the fourth column identifies the sections in which the constant is mentioned. (A boldface section indicates the place the constant is introduced or described.)

Table 3.7. Portals Constants: Other Constants Defined by the Portals API.

Name	Meaning	Base Type	Reference
PTL_ACK_REQ	request an acknowledgment	**ptl_ack_req_t**	**3.13**, 3.13.2
PTL_EQ_HANDLER_NONE	a NULL event queue handler function	**ptl_eq_handler_t**	3.11.6, **3.11.7**
PTL_EQ_NONE	a NULL event queue handle	**ptl_handle_eq_t**	**3.2.2**, 3.10.1, 3.10.6
PTL_EVENT_ACK	acknowledgment event	**ptl_event_kind_t**	**3.11.1**, 3.13.2
PTL_EVENT_GET_END	get event end	**ptl_event_kind_t**	**3.11.1**, 3.13.4
PTL_EVENT_GETPUT_END	getput event end	**ptl_event_kind_t**	**3.11.1**, 3.13.6
PTL_EVENT_GETPUT_START	getput event start	**ptl_event_kind_t**	**3.11.1**, 3.11.3, 3.13.6
PTL_EVENT_GET_START	get event start	**ptl_event_kind_t**	**3.11.1**, 3.13.4
PTL_EVENT_PUT_END	put event end	**ptl_event_kind_t**	**3.11.1**, 3.13.2
PTL_EVENT_PUT_START	put event start	**ptl_event_kind_t**	**3.11.1**, 3.13.2
PTL_EVENT_REPLY_END	reply event end	**ptl_event_kind_t**	**3.11.1**, 3.13.4, 3.13.6
PTL_EVENT_REPLY_START	reply event start	**ptl_event_kind_t**	**3.11.1**, 3.13.4, 3.13.6
PTL_EVENT_SEND_END	send event end	**ptl_event_kind_t**	**3.11.1**, 3.13.2, 3.13.6
PTL_EVENT_SEND_START	send event start	**ptl_event_kind_t**	**3.11.1**, 3.13.2, 3.13.6, 3.11.3
PTL_EVENT_UNLINK	unlink event	**ptl_event_kind_t**	3.10.1, 3.10.5, **3.11.1**
PTL_IFACE_DEFAULT	default interface	**ptl_interface_t**	**3.2.5**
PTL_INS_AFTER	insert after	**ptl_ins_pos_t**	**3.9**, 3.9.2, 3.9.4
PTL_INS_BEFORE	insert before	**ptl_ins_pos_t**	**3.9**, 3.9.2, 3.9.4
PTL_INVALID_HANDLE	invalid handle	**ptl_handle_any_t**	**3.2.2**, 3.14.1
PTL_JID_ANY	wildcard for job identifier	**ptl_jid_t**	3.8, **3.2.6**, 3.12.1
PTL_JID_NONE	job identifiers not supported for process	**ptl_jid_t**	**3.8**
PTL_MD_ACK_DISABLE	a flag to disable acknowledgments	int	**3.10.1**
PTL_MD_EVENT_END_DISABLE	a flag to disable end events	int	**3.10.1, 3.11.3**
PTL_MD_EVENT_START_DISABLE	a flag to disable start events	int	**3.10.1**, 3.11.3
PTL_MD_IOVEC	a flag to enable scatter/gather memory descriptors	int	**3.10.1**, 3.10.2
PTL_MD_MANAGE_REMOTE	a flag to enable the use of remote offsets	int	**3.10.1**, 3.13.2, 3.13.4
PTL_MD_MAX_SIZE	use the *max_size* field in a memory descriptor	unsigned int	**3.10.1**
PTL_MD_OP_GET	a flag to enable *get* operations	int	**3.10.1**, 4.2

continued on next page

continued from previous page

Name	Meaning	Base Type	Reference
PTL_MD_OP_PUT	a flag to enable *put* operations	int	**3.10.1**, 4.2
PTL_MD_THRESH_INF	infinite threshold for a memory descriptor	int	**3.10.1**
PTL_MD_TRUNCATE	a flag to enable truncation of a request	int	**3.10.1**, 4.2
PTL_NID_ANY	wildcard for node identifier fields	**ptl_nid_t**	**3.2.6**, 3.9.2, 3.12.1
PTL_NI_OK	successful event	**ptl_ni_fail_t**	**3.11.4**, 3.11.5
PTL_NO_ACK_REQ	request no acknowledgment	**ptl_ack_req_t**	**3.13**, 3.13.2, 4.1
PTL_PID_ANY	wildcard for process identifier fields	**ptl_pid_t**	**3.2.6**, 3.5.2, 3.9.2, 3.12.1
PTL_PT_INDEX_ANY	wildcard for portal table indexes	**ptl_pt_index_t**	**3.12.1**
PTL_RETAIN	disable unlinking	**ptl_unlink_t**	**3.10.3**
PTL_SR_DROP_COUNT	index for the dropped count register	**ptl_sr_index_t**	**3.2.7**, 3.5.4
PTL_TIME_FOREVER	a flag to indicate unbounded time	**ptl_time_t**	**3.11.12**
PTL_UID_ANY	wildcard for user identifier	**ptl_uid_t**	**3.2.6**, 3.9.2, 3.12.1
PTL_UNLINK	enable unlinking	**ptl_unlink_t**	**3.10.3**

Chapter 4

The Semantics of Message Transmission

The portals API uses four types of messages: *put*, *acknowledgment*, *get*, and *reply*. In this section, we describe the information passed on the wire for each type of message. We also describe how this information is used to process incoming messages.

4.1 Sending Messages

Table 4.1 summarizes the information that is transmitted for a *put* request. The first column provides a descriptive name for the information, the second column provides the type for this information, the third column identifies the source of the information, and the fourth column provides additional notes. Most information that is transmitted is obtained directly from the *put* operation.

IMPLEMENTATION NOTE 25:

Information on the wire

This section describes the information that portals semantics require to be passed between an *initiator* and its *target*. The portals specification does not enforce a given wire protocol or in what order and what manner information is passed along the communication path.

For example, portals semantics require that an *acknowledgment* event contains the memory descriptor from which the *put* originated; i.e., the *acknowledgment* event points to the local memory descriptor that triggered it. This section suggests that the *put* memory descriptor is sent to the *target* and back again in the *acknowledgment* message. If an implementation has another way of identifying the memory descriptor and its event queue, then sending the memory descriptor pointer may not be necessary.

Notice that the handle for the memory descriptor used in the *put* operation is transmitted even though this value cannot be interpreted by the *target*. A value of anything other than `PTL_NO_ACK_REQ` is interpreted as a request for an acknowledgment. In that case the memory descriptor value is sent back to the *initiator* in the *acknowledgment* message. It is needed to find the appropriate event queue and identify the memory descriptor of the original *put*.

A portals header contains 8 bytes of user data. This is useful for out-of-band data transmissions with or without bulk data. The header bytes are stored in the event queue. (See Section 3.10.1 on page 45.)

Table 4.1 is also valid for **PtlPutRegion()** calls. The only difference is that the length information is taken from the function call arguments instead of the memory descriptor to be sent.

Table 4.1. Send Request: Information Passed in a Send Request — **PtlPut()** and **PtlPutRegion()**.

Information	Type	PtlPut() Argument	Notes
operation	int		indicates a *put* request
initiator	**ptl_process_id_t**		local information
user	**ptl_uid_t**		local information
job identifier	**ptl_jid_t**		local information (if supported)
target	**ptl_process_id_t**	*target_id*	
portal index	**ptl_pt_index_t**	*pt_index*	
cookie	**ptl_ac_index_t**	*ac_index*	
match bits	**ptl_match_bits_t**	*match_bits*	
offset	**ptl_size_t**	*remote_offset*	
memory desc	**ptl_handle_md_t**	*md_handle*	no ack, if PTL_NO_ACK_REQ
length	**ptl_size_t**	*md_handle*	*length* member — for **PtlPut()**
length	**ptl_size_t**	*length*	amount of data — for **PtlPutRegion()**
header data	**ptl_hdr_data_t**	*hdr_data*	user data in header
data	*bytes*	*md_handle*	user data

Table 4.2 summarizes the information transmitted in an *acknowledgment*. Most of the information is simply echoed from the *put* request. Notice that the *initiator* and *target* are obtained directly from the *put* request but are swapped in generating the *acknowledgment*. The only new piece of information in the *acknowledgment* is the manipulated length, which is determined as the *put* request is satisfied, and the actual offset used.

IMPLEMENTATION NOTE 26:

Acknowledgment requests

If an *acknowledgment* has been requested, the associated memory descriptor remains in use by the implementation until the *acknowledgment* arrives and can be logged in the event queue. See Section 3.10.5 for how pending operations affect unlinking of memory descriptors.

If the target memory descriptor does not have the PTL_MD_MANAGE_REMOTE flag set, the offset local to the *target* memory descriptor is used. If the flag is set, the offset requested by the *initiator* is used. An *acknowledgment* message returns the actual value used.

Table 4.2. Acknowledgment: Information Passed in an Acknowledgment.

Information	Type	PtlPut() Argument	Notes
operation	int		indicates an *acknowledgment*
initiator	**ptl_process_id_t**	*target_id*	echo *target* of *put*
target	**ptl_process_id_t**	*initiator*	echo *initiator* of *put*
portal index	**ptl_pt_index_t**	*pt_index*	echo
match bits	**ptl_match_bits_t**	*match_bits*	echo
offset	**ptl_size_t**	*remote_offset*	obtained from the operation
memory descriptor	**ptl_handle_md_t**	*md_handle*	echo
requested length	**ptl_size_t**	*md_handle*	echo
manipulated length	**ptl_size_t**		obtained from the operation

Table 4.3 summarizes the information that is transmitted for a *get* request. Like the information transmitted in a *put* request, most of the information transmitted in a *get* request is obtained directly from the **PtlGet()** operation. The memory descriptor must not be unlinked until the *reply* is received.

Table 4.3 needs an additional field for **PtlGetRegion()** calls. The local offset specified by **PtlGetRegion()** needs to be sent with the request and must come back with the reply information.

Table 4.3. Get Request: Information Passed in a Get Request — **PtlGet()** and **PtlGetRegion()**.

Information	Type	PtlGet() Argument	Notes
operation	int		indicates a *get* operation
initiator	**ptl_process_id_t**		local information
user	**ptl_uid_t**		local information
job identifier	**ptl_jid_t**		local information (if supported)
target	**ptl_process_id_t**	*target_id*	
portal index	**ptl_pt_index_t**	*pt_index*	
cookie	**ptl_ac_index_t**	*ac_index*	
match bits	**ptl_match_bits_t**	*match_bits*	
offset	**ptl_size_t**	*remote_offset*	
memory descriptor	**ptl_handle_md_t**	*md_handle*	destination of *reply*
length	**ptl_size_t**	*md_handle*	*length* member
initiator offset	**ptl_size_t**	*local_offset*	for **PtlGetRegion()** only

Table 4.4 summarizes the information transmitted in a *reply*. Like an *acknowledgment*, most of the information is simply echoed from the *get* request. The *initiator* and *target* are obtained directly from the *get* request but are swapped in generating the *reply*. The only new information in the *reply* are the manipulated length, the actual offset used, and the data, which are determined as the *get* request is satisfied.

Table 4.4. Reply: Information Passed in a Reply.

Information	Type	PtlGet() Argument	Notes
operation	int		indicates an *acknowledgment*
initiator	**ptl_process_id_t**	*target_id*	echo *target* of *get*
target	**ptl_process_id_t**	*initiator*	echo *initiator* of *get*
portal index	**ptl_pt_index_t**	*pt_index*	echo
match bits	**ptl_match_bits_t**	*match_bits*	echo
offset	**ptl_size_t**	*remote_offset*	obtained from the operation
memory descriptor	**ptl_handle_md_t**	*md_handle*	echo
requested length	**ptl_size_t**	*md_handle*	echo *length* member
manipulated length	**ptl_size_t**		obtained from the operation
initiator offset	**ptl_size_t**	*local_offset*	for **PtlGetRegion()** only
data	*bytes*		obtained from the operation

Table 4.5 presents the information that needs to be transmitted from the *initiator* to the *target* for a *getput* operation. The result of a *getput* operation is a *reply* as described in Table 4.4.

Table 4.5. Get/Put Request: Information Passed in a Get/Put Request.

Information	Type	PtlGetPut() Argument	Notes
operation	int		indicates a *getput* operation
initiator	**ptl_process_id_t**		local information
user	**ptl_uid_t**		local information
job identifier	**ptl_jid_t**		local information (if supported)
target	**ptl_process_id_t**	*target_id*	
portal index	**ptl_pt_index_t**	*pt_index*	
cookie	**ptl_ac_index_t**	*ac_index*	
match bits	**ptl_match_bits_t**	*match_bits*	
offset	**ptl_size_t**	*remote_offset*	
header data	**ptl_hdr_data_t**	*hdr_data*	user data in header
memory descriptor	**ptl_handle_md_t**	*get_md_handle*	destination of *reply*
length	**ptl_size_t**	*put_md_handle*	*length* member
data	*bytes*	*put_md_handle*	user data

4.2 Receiving Messages

When an incoming message arrives on a network interface, the communication system first checks that the *target* process identified in the request is a valid process that has initialized the network interface (i.e., that the *target* process has a valid portal table). If this test fails, the communication system discards the message and increments the dropped message count for the interface. The remainder of the processing depends on the type of the incoming message. *put*, *get*, and *getput* messages are subject to access control checks and translation (searching a match list), while *acknowledgment* and *reply* messages bypass the access control checks and the translation step.

Acknowledgment messages include a handle for the memory descriptor used in the original **PtlPut()** or **PtlPutRegion()** operation. This memory descriptor will identify the event queue where the event should be recorded. Upon receipt of an acknowledgment, the runtime system only needs to confirm that the memory descriptor and event queue still exist. Should any of these conditions fail, the message is simply discarded, and the dropped message count for the interface is incremented. Otherwise, the system builds an acknowledgment event from the information in the acknowledgment message and adds it to the event queue.

Reception of *reply* messages is also relatively straightforward. Each *reply* message includes a handle for a memory descriptor. If this descriptor exists, it is used to receive the message. A *reply* message will be dropped if the memory descriptor identified in the request does not exist or it has become inactive. In this case, the dropped message count for the interface is incremented. Every memory descriptor accepts and truncates incoming *reply* messages, eliminating the other potential reasons for rejecting a *reply* message.

The critical step in processing an incoming *put*, *get*, or *getput* request involves mapping the request to a memory descriptor. This step starts by using the portal index in the incoming request to identify a list of match entries. This list of match entries is searched in sequential order until a match entry is found whose match criteria matches the match bits in the incoming request and whose memory descriptor accepts the request.

Because *acknowledgment* and *reply* messages are generated in response to requests made by the process receiving these messages, the checks performed by the runtime system for acknowledgments and replies are minimal. In contrast, *put*, *get*, and *getput* messages are generated by remote processes and the checks performed for these messages are more extensive. Incoming *put*, *get*, or *getput* messages may be rejected because:

- the access control index supplied in the request is not a valid access control entry;
- the access control entry identified by the index does not match the identifier of the requesting process;
- the access control entry identified by the access control entry does not match the portal index supplied in the request;
- the portal index supplied in the request is not valid; or
- the match bits supplied in the request do not match any of the match entries with a memory descriptor that accepts the request.

In all cases, if the message is rejected, the incoming message is discarded and the dropped message count for the interface is incremented.

A memory descriptor may reject an incoming request for any of the following reasons:

- the `PTL_MD_OP_PUT` or `PTL_MD_OP_GET` option has not been enabled and the operation is *put*, *get*, or *getput* (Table 4.6); or
- the length specified in the request is too long for the memory descriptor and the `PTL_MD_TRUNCATE` option has not been enabled.

Also see Sections 2.2 and 2.3 and Figure 2.5.

Table 4.6. Portals Operations and Memory Descriptor Flags: A - indicates that the operation will be rejected, and a • indicates that the memory descriptor will accept the operation.

Target Memory Descriptor Flags	Operation		
	put	*get*	*getput*
none	-	-	-
`PTL_MD_OP_PUT`	•	-	-
`PTL_MD_OP_GET`	-	•	-
both	•	•	•

References

Brightwell, R., D. S. Greenberg, A. B. Maccabe, and R. Riesen (2000, February). **Massively Parallel Computing with Commodity Components**. *Parallel Computing 26*, 243–266.

Brightwell, R., T. Hudson, R. Riesen, and A. B. Maccabe (1999, December). **The Portals 3.0 Message Passing Interface**. Technical Report SAND99-2959, Sandia National Laboratories.

Brightwell, R. and L. Shuler (1996, July). **Design and Implementation of MPI on Puma Portals**. In *Proceedings of the Second MPI Developer's Conference*, pp. 18–25.

Compaq, Microsoft, and Intel (1997, December). **Virtual Interface Architecture Specification Version 1.0**. Technical report, Compaq, Microsoft, and Intel.

Cray Research, Inc. (1994, October). **SHMEM Technical Note for C, SG-2516 2.3**. Cray Research, Inc.

Ishikawa, Y., H. Tezuka, and A. Hori (1996). **PM: A High-Performance Communication Library for Multi-user Parallel Envrionments**. Technical Report TR-96015, RWCP.

Lauria, M., S. Pakin, and A. Chien (1998). **Efficient Layering for High Speed Communication: Fast Messages 2.x**. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*.

Maccabe, A. B., K. S. McCurley, R. Riesen, and S. R. Wheat (1994, June). **SUNMOS for the Intel Paragon: A Brief User's Guide**. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pp. 245–251.

Message Passing Interface Forum (1994). **MPI: A Message-Passing Interface standard**. *The International Journal of Supercomputer Applications and High Performance Computing 8*, 159–416.

Message Passing Interface Forum (1997, July). **MPI-2: Extensions to the Message-Passing Interface**. Message Passing Interface Forum.

Myricom, Inc. (1997). **The GM Message Passing System**. Technical report, Myricom, Inc.

Riesen, R., R. Brightwell, and A. B. Maccabe (2005). **The Evolution of Portals, an API for High Performance Communication**. *To be published*.

Riesen, R. and A. B. Maccabe (2002, November). **RMPP: The Reliable Message Passing Protocol**. In *Workshop on High-Speed Local Networks HSLN'02*, Tampa, Florida.

Sandia National Laboratories (1996). **ASCI Red**. Sandia National Laboratories. http://www.sandia.gov/ASCI/TFLOP.

Shuler, L., C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup (1995). **The Puma Operating System for Massively Parallel Computers**. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group.

Task Group of Technical Committee T11 (1998, July). **Information Technology - Scheduled Transfer Protocol - Working Draft 2.0**. Technical report, Accredited Standards Committee NCITS.

Appendix A

Frequently Asked Questions

This document is a specification for the portals 3.3 API. People using and implementing Portals sometimes have questions that the specification does not address. In this appendix we answer some of the more common questions.

Q Are Portals a wire protocol?

A No. The portals document defines an API with semantics that specify how messages move from one address space to another. It does not specify how the individual bytes are transferred. In that sense it is similar to the socket API: TCP/IP or some other protocol is used to reliably transfer the data. Portals assume an underlying transport mechanism that is reliable and scalable.

Q How are Portals different from the sockets API (TCP/IP) or the MPI API?

A Sockets are stream-based while Portals are message-based. Portals implementations can use the a priori knowledge of the total message length to manage the buffers and protocols to be used. The portals API makes it easy to let the implementation know in advance where in user space incoming data should be deposited. The sockets API makes this more difficult because the implementation will not know where data has to go until the application issues a *read()* request.

The sockets API using TCP/IP is connection-oriented which limits scalability because state has to be maintained for each open connection and the number of connections increases with the size of the machine.

MPI is a higher level API than Portals. For example, the function **MPIrecv()** can be issued before or after the sender has sent the message matching this receive. The MPI implementation has to take care of buffering the message if the receive has not been posted yet. Portals simply discard messages for which the receiver is not yet ready.

Portals are ideally suited to be used by an MPI implementation. An application programmer, however, may grow frustrated by Portals' lack of user-friendliness. We recommend that Portals be used by systems programmers and library writers, not application programmers.

Q What about GM, FM, AM, PM, etc.?

A There are many communication paradigms, and, especially in the early 1990s, many experiments were conducted on how to best pass messages among supercomputer nodes; hence, the proliferation of the various *M message passing layers.

Some of them, such as GM, are hardware specific. Almost every network interface vendor has its own API to access its hardware. Portals are portable and open source. They were designed to run on a wide variety of networks with NICs that are programmable or not. This was an important design criteria for Portals 3.0 when work on Cplant started.

Most of the research message passing layers do not provide reliability because they were designed for networks that are, for all practical purposes, reliable. While Portals themselves do not provide a wire protocol, Portals demand that the transport layer underneath is reliable. This places Portals a level above the other APIs in the networking stack. On reliable networks, such as ASCI Red, Portals can be implemented

without a wire protocol. On unreliable networks, such as Myrinet, Portals can run over GM or some other protocol that implements reliability.

Some of the research paradigms do not scale to thousands of nodes. In order to control local resources, some of them use send tokens to limit the number of messages that can be sent through the network at any given time. As a machine and its network grow, this imposes severe limitations and degrades the scalability of the message passing layer.

Q What is a NAL?

A NAL stands for Network Abstraction Layer. All current portals 3.x implementations are in some way or another derived from the reference implementation which employs a NAL. A NAL is a very nice way to abstract the network interface from a portals library. The library implements common portals functions in user space and can be easily ported from one architecture to another. On the other side of the NAL, in protected space, we find routines that are more specific to a given architecture and network interface.

Q Must Portals be implemented using a NAL?

A No. A NAL provides a nice abstraction and makes it easier to port portals implementations, but the API and semantics of Portals do not require a NAL.

Q Why does the portals API not specify a barrier operation?

A Earlier versions of the API had a barrier function. It turned out to be quite difficult to implement on some architectures. The main problem was that nodes would boot in intervals and not be ready to participate in a portals barrier operation until later. The portals implementations had to rely on the runtime system to learn when nodes became active. The runtime systems, in turn, usually had some form of barrier operation that allowed them to synchronize nodes after booting or after job load.

Because that functionality already existed and it made portals implementations difficult, we decided to eliminate the barrier operation from the portals API. However, future versions of Portals may have collective operations. In that case, the portals barrier would be re-introduced.

Appendix B

Portals Design Guidelines

Early versions of Portals were based on the idea to use data structures to describe to the transport mechanism how data should be delivered. This worked well for the Puma OS on the Intel Paragon but not so well under Linux on Cplant. The solution was to create a thin API over those data structures and add a level of abstraction. The result was Portals 3.x.

When designing and expanding this API, we were guided by several principles and requirements. We have divided them into three categories: requirements that must be fulfilled by the API and its implementations, requirements that should be met, and a wish list of things that would be nice if Portals 3.x could provide them.

B.1 Mandatory Requirements

Message passing protocols. Portals *must* support efficient implementations of commonly used message passing protocols.

Portability. It *must* be possible to develop implementations of Portals on a variety of existing message passing interfaces.

Scalability. It *must* be possible to write efficient implementations of Portals for systems with thousands of nodes.

Performance. It *must* be possible to write high performance (e.g., low latency, high bandwidth) implementations of Portals on existing hardware.

Multiprocess support. Portals *must* support use of the communication interface by at least two processes per CPU per node.

Communication between processes from different executables. Portals *must* support the ability to pass messages between processes instantiated from different executables.

Runtime independence. The ability of a process to perform message passing *must not* depend on the existence of an external runtime environment, scheduling mechanism, or other special utilities outside of normal UNIX process startup.

Memory protection. Portals *must* ensure that a process cannot access the memory of another process without consent.

B.2 The *Will* Requirements

Operational API. Portals *will* be defined by operations, not modifications to data structures. This means that the interface will have explicit operations to send and receive messages. (It does not mean that the receive operation will involve a copy of the message body.)

MPI. It *will* be possible to write an efficient implementation of the point-to-point operations in MPI 1 using Portals.

Myrinet. It *will* be possible to write an efficient implementation of Portals using Linux as the host OS and Myrinet interface cards.

Sockets Implementation. It *will* be possible to write an implementation of Portals based on the sockets API.

Message Size. Portals *will not* impose an arbitrary restriction on the size of message that can be sent.

OS bypass. Portals *will* support an OS bypass message passing strategy. That is, high performance implementations of the message passing mechanisms will be able to bypass the OS and deliver messages directly to the application.

Put/Get. Portals *will* support remote put/get operations.

Packets. It *will* be possible to write efficient implementations of Portals that packetize message transmission.

Receive operation. The receive operation of Portals *will* use an address and length pair to specify where the message body should be placed.

Receiver managed communication. Portals *will* support receive-side management of message space, and this management will be performed during message receipt.

Sender managed communication. Portals *will* support send-side management of message space.

Parallel I/O. Portals *will* be able to serve as the transport mechanism for a parallel file I/O system.

Gateways. It *will* be possible to write *gateway* processes using Portals. A gateway process is a process that receives messages from one implementation of Portals and transmits them to another implementation of Portals.

Asynchronous operations. Portals *will* support asynchronous operations to allow computation and communication to overlap.

Receive side matching. Portals *will* allow matching on the receive side before data is delivered into the user buffer.

B.3 The *Should* Requirements

Message Alignment. Portals *should* not impose any restrictions regarding the alignment of the address(es) used to specify the contents of a message.

Striping. Portals *should* be able to take advantage of multiple interfaces on a single logical network to improve the bandwidth

Socket API. Portals *should* support an efficient implementation of sockets (including UDP and TCP/IP).

Scheduled Transfer. It *should* be possible to write an efficient implementation of Portals based on Scheduled Transfer (ST).

Virtual Interface Architecture. It *should* be possible to write an efficient implementation of Portals based on the Virtual Interface Architecture (VIA).

Internetwork consistency. Portals *should not* impose any consistency requirements across multiple networks/interfaces. In particular, there will not be any memory consistency/coherency requirements when messages arrive on independent paths.

Ease of use. Programming with Portals *should* be no more complex than programming traditional message passing environments such as UNIX sockets or MPI. An in-depth understanding of the implementation or access to implementation-level information should not be required.

Minimal API. Only the smallest number of functions and definitions necessary to manipulate the data structures should be specified. That means, for example, that convenience functions, which can be implemented with the already defined functions, will not become part of the API.

One exception to this is if a non-native implementation would suffer in scalability or take a large performance penalty.

Appendix C

A README Template

Each portals implementation should provide a README file that details implementation-specific choices. This appendix serves as a template for such a file by listing which parameters should be specified.

Limits. The call **PtlNIInit()** accepts a desired set of limits and returns a set of actual limits. The README should state the possible ranges of actual limits for this implementation, as well as the acceptable ranges for the values passed into **PtlNIInit()**. See Section 3.5.1

Status Registers. Portals define a set of status registers (Section 3.2.7). The type **ptl_sr_index_t** defines the mandatory `PTL_SR_DROP_COUNT` and all other, implementation specific indexes. The README should list what indexes are available and what their purposes are.

Network interfaces. Each portals implementation defines `PTL_IFACE_DEFAULT` to access the default network interface on a system (Sections 3.2.5 and 3.5.2). An implementation that supports multiple interfaces must specify the constants used to access the various interfaces through **PtlNIInit()**.

Portal table. The portals specification says that a compliant implementation must provide at least 8 entries per portal table (Section 3.5). The README file should state how many entries will actually be provided.

Maximum size of a PtlGetPut(). The **PtlGetPut()** operation (Sections 3.5.1 and 3.13.6) supports at least 8 bytes of data. An implementation may provide more. The actual value should be mentioned in the README file. It can also be retrieved by the user with the call **PtlNIInit()** (Sections 3.5.2 and 3.5.1).

Distance measure. An implementation should state in its README what measure, if any, is returned by **PtlNIDist()** (Section 3.5.5).

Job identifiers. The README file should indicate whether the implementation supports job identifiers (Section 3.8).

Alignment. If an implementation favors specific alignments for memory descriptors, the README should state what they are and the (performance) consequences if they are not observed (Section 3.10.1).

Event handlers. The README should state implementation-specific restrictions on event handlers (Section 3.11.8).

Appendix D

Implementations

In this appendix we briefly mention two portals 3.3 implementations: A reference implementation and one that runs on Cray's XT3 Red Storm.

D.1 Reference Implementation

A portals 3.3 reference implementation has been written and is maintained by Jim Schutt. The main goal of the reference implementation is to provide a working example that implements the syntax, semantics, and spirit of Portals 3.3 as described in this document.

The reference implementation uses the NAL (Network Abstraction Layer) concept to separate the network independent part from the code that is specific to the API and protocols of the underlying layer. The reference implementation uses the sockets API and TCP/IP for its transport mechanism. While this is not overly efficient, the code used to implement Portals 3.3 can be understood by the many people who are familiar with the sockets API. Furthermore, TCP/IP is so widespread that the reference implementation is executable on a large array of machines and networks.

There is a build option that disables a separate progress thread which allows Portals to make progress (sending an *acknowledgment* for example) without the layer above making calls into the portals library. This speeds up the implementation but violates the progress rule.

The source code for the implementation is freely available from the following site:

`ftp://ftp.sandia.gov/outgoing/pub/portals3`

In addition to comments in the code, it contains several README files that describe the implementation. Feedback is highly encouraged to the code author, `jaschut@sandia.gov`, and the Portals 3.3 team at Sandia National Laboratories, `p3@sandia.gov`.

A NAL that runs in Linux kernel space is currently under development.

We maintain a portals web site at `http://www.cs.sandia.gov/Portals` with links to the latest reference implementation and other information.

D.2 Portals 3.3 on the Cray XT3 Red Storm

There are two implementations of Portals available on Cray's XT3 Red Storm system. One, generic, is provided by Cray with the machine. The second, accelerated, is under active development at Sandia National Laboratories. There are plans to merge the two versions in the future.

D.2.1 Generic

This is the version provided by Cray with its XT3 Red Storm systems. A large portion of the portals code is implemented inside the kernel. When messages arrive at the Seastar NIC, it causes an interrupt and lets the kernel process the portals header; i.e., resolve portal table addressing and match list traversal. The accelerated version under development places more of the portals code inside the Seastar NIC and avoids the interrupt processing on each message arrival.

The generic implementation does not completely match the definitions in this document. The main differences are listed here:

- **PtlNIDist()** is not implemented.
- **PtlACEntry()** is not implemented. Calling it has no effect.
- **PtlHandleIsEqual()** is not implemented.
- **Limitations on IOVECs:** Only the first and last entry can be unaligned (at the head of the buffer and at the tail of the buffer, everything else must be quad-byte aligned).
- There are three new functions that are not part of this document: **PtlIsValidHandle()**, **PtlSetInvalidHandle()**, and **PtlTestGetPut()**.
- The following return codes are not implemented: **PTL_AC_INDEX_INVALID**, **PTL_MD_ILLEGAL**, and **PTL_IFACE_INVALID**.
- The type **ptl_size_t** is 32 bits wide, not 64 bits.
- **PtlEQGet()** and **PtlEQWait()** may return a **ptl_event_t** structure that is not fully filled in.
- Event queue handlers may block, make system calls, perform I/O, can call the portals event queue functions, and may initiate *put* and *get* operations.

Please refer to Cray documentation for up-to-date information.

D.2.2 Accelerated

An accelerated version that avoids interrupts for each message arrival is being developed and tested at Sandia National Laboratories. At the moment is has more limitations than the generic implementation and leaves out several features discussed in this document. The main ones are:

- Event handlers are not supported.
- Adds a **PtlPost()** call which combines a **PtlMEInsert()** and **PtlMDUpdate()** call. This eliminates a protection domain boundary crossing in many of the common usage cases.
- The **PtlGet()** operation generates `PTL_EVENT_SEND_START` and `PTL_EVENT_SEND_END` events.

Since this implementation is still under active development, further changes are to be expected.

Appendix E

Summary of Changes

The first version of this document described Portals version 3.0 [Brightwell et al. 1999]. Since then we have made changes to the API and semantics of Portals, as well as changes to the document. This appendix summarizes the changes between the individual versions and outlines the path to the current 3.3 version.

E.1 Changes From Version 3.0 to 3.1

E.1.1 Thread Issues

The most significant change to the interface from version 3.0 to 3.1 involves the clarification of how the interface interacts with multi-threaded applications. We adopted a generic thread model in which processes define an address space and threads share the address space. Consideration of the API in the light of threads led to several clarifications throughout the document:

1. Glossary:
   (a) added a definition for *thread*, and
   (b) reworded the definition for *process*.
2. Section 2: added Section 2.4 to describe the multi-threading model used by the portals API.
3. Section 3.4.1: **PtlInit()** must be called at least once and may be called any number of times.
4. Section 3.4.2: **PtlFini()** should be called once as the process is terminating and not as each thread terminates.
5. Section 3.7: Portals does not define thread identifiers.
6. Section 3.5: network interfaces are associated with processes, not threads.
7. Section 3.5.2: **PtlNIInit()** must be called at least once and may be called any number of times.
8. Section 3.11.10: **PtlEQGet()** returns **PTL_EQ_EMPTY** if a thread is blocked on **PtlEQWait()**.
9. Section 3.11.11: waiting threads are awakened in FIFO order.

Two functions, **PtlNIBarrier()** and **PtlEQCount()**, were removed from the API. **PtlNIBarrier()** was defined to block the calling process until all of the processes in the application group had invoked **PtlNIBarrier()**. We now consider this functionality, along with the concept of groups (see the discussion under "other changes") to be part of the runtime system, not part of the portals API. **PtlEQCount()** was defined to return the number of events in an event queue. Because external operations may lead to new events being added and other threads may remove events, the value returned by **PtlEQCount()** would have to be a hint about the number of events in the event queue.

E.1.2 Handling Small, Unexpected Messages

Another set of changes relates to handling small unexpected messages in MPI. In designing version 3.0, we assumed that each unexpected message would be placed in a unique memory descriptor. To avoid the need to process a long list of memory descriptors, we moved the memory descriptors out of the match list and hung them off of a single match list entry. In this way, large unexpected messages would only encounter a single "short message" match list entry before encountering the "long message" match list entry. Experience with this strategy identified resource management problems with this approach. In particular, a long sequence of very short (or zero length) messages could quickly exhaust the memory descriptors constructed for handling unexpected messages. Our new strategy involves the use of several very large memory descriptors for small unexpected messages. Consecutive unexpected messages will be written into the first of these memory descriptors until the memory descriptor fills up. When the first of the "small memory" descriptors fills up, it will be unlinked and subsequent short messages will be written into the next "short message" memory descriptor. In this case, a "short message" memory descriptor will be declared full when it does not have sufficient space for the largest possible unexpected message that is considered small.

This led to two significant changes. First, each match list entry now has a single memory descriptor rather than a list of memory descriptors. Second, in addition to exceeding the operation threshold, a memory descriptor can be unlinked when the local offset exceeds a specified value. These changes have led to several changes in this document:

1. Section 2.2:
   (a) removed references to the memory descriptor list, and
   (b) changed the portals address translation description to indicate that unlinking a memory descriptor implies unlinking the associated match list entry–match list entries can no longer be unlinked independently from the memory descriptor.
2. Section 3.9.2:
   (a) removed unlink from argument list,
   (b) removed description of `PTL_UNLINK` type, and
   (c) changed wording of the error condition when the portal table index already has an associated match list.
3. Section 3.9.4: removed unlink from argument list.
4. Section 3.10.1: added *max_offset*.
5. Section 3.10.3:
   (a) added description of `PTL_UNLINK` type,
   (b) removed reference to memory descriptor lists,
   (c) changed wording of the error condition when match list entry already has an associated memory descriptor, and
   (d) changed the description of the *unlink* argument.
6. Section 3.10: removed **PtlMDInsert()** operation.
7. Section 3.10.4: removed references to memory descriptor list.
8. Section 3.10.5: removed reference to memory descriptor list.
9. Section 3.15: removed references to PtlMDInsert.
10. Section 4: removed reference to memory descriptor list.

11. Revised the MPI example to reflect the changes to the interface.

Several changes have been made to improve the general documentation of the interface.

1. Section 3.2.2: documented the special value `PTL_EQ_NONE`.
2. Section 3.2.6: documented the special value `PTL_ID_ANY`.
3. Section 3.10.4: documented the return value **PTL_EQ_INVALID**.
4. Section 3.10.6: clarified the description of the **PtlMDUpdate()** function.
5. Introduced a new section to document the implementation defined values.
6. Section 3.15: modified Table 3.7 to indicate where each constant is introduced and where it is used.

E.1.3 Other Changes

E.1.3.1 Implementation Defined Limits (Section 3.5.2)

The earlier version provided implementation defined limits for the maximum number of match entries, the maximum number of memory descriptors, etc. Rather than spanning the entire implementation, these limits are now associated with individual network interfaces.

E.1.3.2 Added User Identifiers (Section 3.6)

Group identifiers had been used to simplify access control entries. In particular, a process could allow access for all of the processes in a group. User identifiers have been introduced to regain this functionality. We use user identifiers to fill this role.

E.1.3.3 Removed Group Identifiers and Rank Identifiers (Section 3.7)

The earlier version of Portals had two forms for addressing processes: ¡node identifier, process identifier¿ and ¡group identifier, rank identifier¿. A process group was defined as the collection processes created during application launch. Each process in the group was given a unique rank in the range 0 to $n-1$, where n was the number of processes in the group. We removed groups because they are better handled in the runtime system.

E.1.3.4 Match Lists (Section 3.9.2)

It is no longer illegal to have an existing match entry when calling **PtlMEAttach()**. A position argument was added to the list of arguments supplied to **PtlMEAttach()** to specify whether the new match entry is prepended or appended to the existing list. If there is no existing match list, the position argument is ignored.

E.1.3.5 Unlinking Memory Descriptors (Section 3.10)

Previously, a memory descriptor could be unlinked if the offset exceeded a threshold upon the completion of an operation. In this version, the unlinking is delayed until there is a matching operation that requires more memory than is currently available in the descriptor. In addition to changes in Section 3.10, this led to a revision of Figure 2.5.

E.1.3.6 Split Phase Operations and Events (Section 3.11)

Previously, there were five types of events: PTL_EVENT_PUT, PTL_EVENT_GET, PTL_EVENT_REPLY, PTL_EVENT_SENT, and PTL_EVENT_ACK. The first four of these reflected the completion of potentially long operations. We have introduced new event types to reflect the fact that long operations have a distinct starting point and a distinct completion point. Moreover, the completion may be successful or unsuccessful.

In addition to providing a mechanism for reporting failure to higher levels of software, this split provides an opportunity for for improved ordering semantics. Previously, if one process initiated two operations (e.g., two *put* operations) on a remote process, these operations were guaranteed to complete in the same order that they were initiated. Now, we only guarantee that the initiation events are delivered in the same order. In particular, the operations do not need to complete in the order that they were initiated.

E.1.3.7 Well Known Process Identifiers (Section 3.5.2)

To support the notion of "well known process identifiers," we added a process identifier argument to the arguments for **PtlNIInit()**.

E.2 Changes From Version 3.1 to 3.2

1. Updated version number to 3.2 throughout the document
2. Section 3.7.2: added **PTL_SEGV** to error list for **PtlGetId()**.
3. Section 3.9.2: added **PTL_ME_LIST_TOO_LONG** to error list for **PtlMEAttach()**.
4. Section 3.9.5: removed text referring to a list of associated memory descriptors.
5. Section 3.10.5: added text to describe unlinking a free-floating memory descriptor.
6. Table 3.4: added entry for **ptl_seq_t**.
7. Section 3.10.1:
   (a) added definition of *max_offset*.
   (b) added text to clarify PTL_MD_MANAGE_REMOTE.
8. Section 3.10.3: modified text for *unlink_op*.
9. Section 3.5.2: added text to clarify multiple calls to **PtlNIInit()**.
10. Section 3.10.3: added text to clarify *unlink_nofit*.
11. Section 4.2: removed text indicating that a memory descriptor will reject a message if the associated event queue is full.
12. Section 3.10.5: added **PTL_MD_IN_USE** error code and text to indicate that only memory descriptors with no pending operations can be unlinked.
13. Table 3.6: added **PTL_MD_IN_USE** return code.
14. Section 3.11.5: added user identifier field, memory descriptor handle field, and NI specific failure field to the **ptl_event_t** structure.
15. Table 3.4: added **ptl_ni_fail_t**.

16. Section 3.11.5: added PTL_EVENT_UNLINK event type.
17. Table 3.5: removed **PtlTransId()**.
18. Section 3.9.2, Section 3.9.4, and Section 3.13.2: listed allowable constants with relevant fields.
19. Table 3.5: added **PtlMEAttachAny()** function.
20. Table 3.6: added **PTL_PT_FULL** return code for **PtlMEAttachAny()**.
21. Table 3.7: updated to reflect new event types.
22. Section 3.2.6: added **ptl_nid_t**, **ptl_pid_t**, and **ptl_uid_t**.
23. Section 3.5.1: added *max_md_iovecs* and *max_me_list*.
24. Section 3.10: changed *max_offset* to *max_size* and added PTL_MD_IOVEC option.
25. Added Section 3.8.
26. Added Section 3.13.6.
27. Deleted the chapter with obsolete examples.

E.3 Changes From Version 3.2 to 3.3

E.3.1 API Changes

1. Section 3.11.12: added **PtlEQPoll()**.
2. Section 3.13.3: added **PtlPutRegion()**.
3. Section 3.13.5: added **PtlGetRegion()**.
4. Section 3.10: added PTL_MD_EVENT_START_DISABLE and PTL_MD_EVENT_END_DISABLE options.
5. Section 3.11.6: added event queue handler capability.
6. Revised naming scheme to be consistent across the entire API.

E.3.2 Semantic Clarifications

Updating the specification and providing better descriptions for some items may have invalidated the semantics of earlier implementations because the earlier documentation was vague or missing information. In this section we document these clarifications.

1. Deleted **PTL_IFACE_DUP**. Interfaces can be initialized several times (by threads).
2. The *remote_offset* in an *acknowledgment* operation now reflects the value used on the remote memory descriptor. By default, it is the local offset, not the offset requested by the *put* operation, unless the remote memory descriptor has the PTL_MD_MANAGE_REMOTE flag set.
3. Ignore the PTL_MD_MAX_SIZE option of a memory descriptor if PTL_MD_MANAGE_REMOTE is set (Section 3.10.1).

E.3.3 Document Changes

1. Converted the Lyx document to LaTeX.
2. Bumped document revision number to 2.x.
3. Formatted the document according to the Sandia Technical report guidelines.
4. Moved change summaries to the end of the document.
5. Used color and C language specific syntax highlighting in listings.
6. Used macros extensively for greater consistency and error checking.
7. Moved return codes to the end of a definition.
8. Corrected a number of mistyped identifiers (e.g., plt_ instead of ptl_).
9. Changed bibliography style to Chicago Style.
10. Added an index.
11. Made small editorial changes.
12. Added page headers.
13. Added a citation for PM, since we mention it.
14. Added a section on frequently asked questions.
15. Added a section about portals design guidelines.
16. Changed timeout in **PtlEQPoll()** to **ptl_time_t**.
17. Added **ptl_md_iovec_t** , **ptl_ni_limits_t** , and **ptl_time_t** to Table 3.4, sorted it alphabetically, and removed **PtlMDAlloc()**.
18. Added **PtlGetUid()** and **PtlHandleIsEqual()** to Table 3.5, sorted it alphabetically, and made column headings boldface.
19. Added a preface
20. Changed **PtlWait()** to **PtlEQWait()** in Table 3.6 and sorted it alphabetically.
21. Changed **PTL_NO_UPDATE** to **PTL_MD_NO_UPDATE** in Section 3.10.6.
22. Reformatted Table 3.7 to fit within the width of the page, sorted it alphabetically, and corrected some of the references. Corrected `PTL_INSERT_AFTER` to be `PTL_INS_AFTER`, and `PTL_INSERT_BEFORE` to be `PTL_INS_BEFORE`. Added `PTL_MD_MAX_SIZE` and `PTL_NI_OK`.
23. Replaced occurrences of `PTL_MD_NONE` with `PTL_NO_ACK_REQ`.
24. Corrected many small inconsistencies (e.g., Table 3.7 contained both `PTL_EVENT_ACK_START` and `PTL_EVENT_ACK_END`, instead of only `PTL_EVENT_ACK`).
25. Section 3.11.1 only described some of the events (and had the total number of them wrong).
26. Added the following paragraph to Section 3.10.1:

    > If both `PTL_MD_EVENT_START_DISABLE` and `PTL_MD_EVENT_END_DISABLE` are specified, no events will be generated. This includes `PTL_EVENT_UNLINK` and `PTL_EVENT_ACK`. If neither `PTL_MD_EVENT_START_DISABLE` nor `PTL_MD_EVENT_END_DISABLE`, or only one of them is specified, then `PTL_EVENT_UNLINK` and `PTL_EVENT_ACK` events will be generated.

27. Added start event processing to Figure 2.5.

28. Added *getput* where needed.

29. Made numerous clarifications in text.

30. Added implementation notes and a list of them to the front matter.

31. Added new figures for the basic portals operations in Section 2.1; expanded descriptions of those figures.

32. Added diagrams to show event types and sequence (Figure 3.1).

33. Added Table 4.5 to include information contained in a *getput* request.

34. Added wire information for *getput* requests in Section 4.1, corrected the information that was incorrect, commented on **PtlPut()** and **PtlGet()**, and added clarifications.

35. Consolidated argument name *jobid* to *jid*.

36. Changed references to page and section numbers into active links when viewed using Adobe's Acrobat Reader.

Index

A

A

ac_index (field) 61–67, 76–78
accelerated 92
access control **24**, 28, 30, 31, 36, **61**
access control entry 61
access control table 61
ack_req (field) 63, 64
acknowledgment . . . . . *see* operations
acknowledgment type **62**
actual (field) 32, 33
address space opening 21
address translation **21**, 23, 26, 78, 94
addressing, portals 26
alignment 43, 89
API . . . . . 12, [13]
API summary **68**
application bypass 16, 18, **18**, 21
application space 23
argument names . . . . . *see* structure fields
ASCI . . . . . [13]
atomic swap . . . . . *see* swap
atomic update 48
authors
  Compaq, Microsoft, and Intel . . 16, 18, (81)
  Message Passing Interface Forum . . 16, (81)
  Myricom, Inc. 18, (81)
  Task Group of Technical Committee T11 18, 21, (81)
  Brightwell and Shuler 17
  Brightwell et al. 12, 16, 93
  Brightwell, Ron . . . . . (81)
  Chien, Andrew . . . . . (81)
  Cray Research, Inc. 21
  Fisk, Lee Ann . . . . . (81)
  Greenberg, David S. . . . . (81)
  Hori, A. . . . . (81)
  Hudson, Tramm . . . . . (81)
  Ishikawa et al. 13, 18
  Ishikawa, Y. . . . . (81)
  Jong, Chu . . . . . (81)
  Lauria et al. 18
  Lauria, Mario . . . . . (81)
  Maccabe et al. 16
  Maccabe, Arthur B. . . . . (81)
  McCurley, Kevin S. . . . . (81)
  Message Passing Interface Forum 21
  Pakin, Scott . . . . . (81)
  Riesen and Maccabe 17
  Riesen et al. 17
  Riesen, Rolf . . . . . (81)
  Sandia National Laboratories 17
  Shuler et al. 16
  Shuler, Lance . . . . . (81)
  Stallcup, T. Mack . . . . . (81)
  Tezuka, H. . . . . (81)
  van Dresser, David . . . . . (81)
  Wheat, Stephen R. . . . . (81)

B

background **16**
barrier operation 84, 93
Barsis, Ed 4
Barton, Eric 4
base (field) 41, 42
Braam, Peter 4
Brightwell, Ron 1, 3
buffer alignment 43, 89
bypass
  application 16, 18, **18**, 21
  OS 16, **18**, 86

C

Camp, Bill 4
changes, API and document 93
collective operations 84
communication model **17**
conditional update 49
connection-oriented 16, 83
connectionless 16, 17
constants
  PTL_ACK_REQ 27, 62, 63, 72
  PTL_EQ_HANDLER_NONE 56, 57, 72
  PTL_EQ_NONE . . 28, 45, 48, 49, 51, 72, 95
  PTL_EVENT_ACK . . 45, 51, 53, 62, 63, 72, 96, 98
  PTL_EVENT_ACK_END 98
  PTL_EVENT_ACK_START 98
  PTL_EVENT_GET 96
  PTL_EVENT_GET_END 50, 53, 65, 72
  PTL_EVENT_GET_START . . 50, 53, 65, 72
  PTL_EVENT_GETPUT_END 51, 53, 66, 72
  PTL_EVENT_GETPUT_START . 51, 53, 66, 72
  PTL_EVENT_PUT 96
  PTL_EVENT_PUT_END 50, 53, 62, 72
  PTL_EVENT_PUT_START . . 50, 53, 62, 72
  PTL_EVENT_REPLY 96
  PTL_EVENT_REPLY_END . 51, 53, 65, 66, 72

C

PTL_EVENT_REPLY_START . . 51–53, 65, 66, 72
PTL_EVENT_SEND_END . . 51–53, 62, 63, 66, 72, 92
PTL_EVENT_SEND_START 51, 53, 62, 63, 66, 72, 92
PTL_EVENT_SENT 96
PTL_EVENT_UNLINK . 42, 45, 47, 51, 53, 55, 72, 97, 98
PTL_IFACE_DEFAULT 28, 72, 89
PTL_INS_AFTER 39, 41, 72, 98
PTL_INS_BEFORE 39, 41, 72, 98
PTL_INVALID_HANDLE 28, 68, 72
PTL_JID_ANY 29, 61, 72
PTL_JID_NONE 38, 54, 72
PTL_MD_ACK_DISABLE 44, 72
PTL_MD_EVENT_END_DISABLE . 44, 45, 51, 53, 72, 97, 98
PTL_MD_EVENT_START_DISABLE . . 44, 45, 51, 53, 72, 97, 98
PTL_MD_IOVEC 44, 45, 72, 97
PTL_MD_MANAGE_REMOTE . 44, 63, 65, 67, 72, 76, 96, 97
PTL_MD_MAX_SIZE 44, 72, 97, 98
PTL_MD_NONE 98
PTL_MD_OP_GET 44, 67, 72, 79
PTL_MD_OP_PUT 44, 67, 73, 79
PTL_MD_THRESH_INF 43, 73
PTL_MD_TRUNCATE 44, 73, 79
PTL_NI_OK 53, 55, 73, 98
PTL_NID_ANY 29, 39, 61, 73
PTL_NO_ACK_REQ . 62, 63, 73, 75, 76, 98
PTL_PID_ANY 29, 32, 39, 61, 73
PTL_PT_INDEX_ANY 61, 73
PTL_RETAIN 38, 39, 46, 73
PTL_SR_DROP_COUNT 29, 34, 73, 89
PTL_TIME_FOREVER 60, 73
PTL_UID_ANY 29, 61, 73
PTL_UNLINK 38, 39, 46, 73, 94
PTL_ID_ANY 95
PTL_INSERT_AFTER 98
PTL_INSERT_BEFORE 98
summary 71
cookie 24
count (field) 56
Cplant 12
CPU interrupts 18
Cray XT3 91

D

data movement **21**, 26, 57, **62**
data types 27, 69
denial of service 25
design guidelines 85
desired (field) 32
discarded events 62
discarded messages 17, 21, 23, 78, 79, 83
distance 34, 89
distance (field) 34, 35
DMA . . . . . [13]
dropped message count 73, 78, 79
dropped messages 29, 58–60, 71

E

eq_handle (field) 45–49, 56–59
eq_handler (field) 56, 57
eq_handles (field) 60
event **50**
disable 44, 45, 51, 53, 72, 97, 98
failure notification 53
handler 50, 55, 56, 69, 72, 89
occurrence 51
order 53
semantics 60, 96
start/end . . 23, 44, 50, 51, 53, 55, 60, 72, 99
types 50, 52
types (diagram) 52
event (field) 58–60
event queue . . . . . [13]
allocation 55
freeing 57
get 58
handler 55, 56, 58, 59
order 53
poll 59
type **54**
wait 58
event threshold . . . . . *see* threshold
events 18

F

failure notification 53
FAQ 83
faults **18**
Fisk, Lee Ann 4
flow control
user-level 16
function return codes . . . . . *see* return codes
functions
MPIrecv 83
PtlACEntry **61**, 69–71, 92
PtlEQAlloc 27, 50, 55, **55**, 57, 69–71
PtlEQCount 93
PtlEQFree 50, **57**, 69–71
PtlEQGet . 50, 56–58, **58**, 59–61, 69–71, 92, 93
PtlEQPoll 26, 50, 56–59, **59**, 60, 61, 69, 70, 97, 98

PtlEQWait 26, 50, 56–58, **58**, 59–61, 69–71, 92, 93, 98
PtlFini 29, 30, **30**, 70, 71, 93
PtlGet 57, 62, **65**, 66, 69–71, 77, 92, 99
PtlGetId 33, 36, **37**, 38, 69, 70, 96
PtlGetJid **38**, 69, 70
PtlGetPut . 31, 57, 62, 66, **67**, 69, 70, 78, 89
PtlGetRegion . 57, 62, 65, **66**, 69, 70, 77, 97
PtlGetUid **36**, 70, 98
PtlHandleIsEqual **68**, 69, 70, 92, 98
PtlInit 27, **29**, 70, 71, 93
PtlIsValidHandle 92
PtlMDAlloc 98
PtlMDAttach 42, **45**, 46, 47, 69–71
PtlMDBind 42, **47**, 48, 69–71
PtlMDInsert 94
PtlMDUnlink 43, 47, 48, **48**, 50, 69–71
PtlMDUpdate . 43, 46, 48, **48**, 69–71, 92, 95
PtlMEAttach . 30, 38, 39, **39**, 40, 41, 69–71, 95, 96
PtlMEAttachAny . 30, 38, 40, **40**, 69–71, 97
PtlMEInsert 38, **41**, 69–71, 92
PtlMEUnlink 38, **42**, 69–71
PtlNIBarrier 93
PtlNIDist 31, **34**, 35, 69–71, 89, 92
PtlNIFini 30, **33**, 69–71
PtlNIHandle 31, **35**, 69–71
PtlNIInit . 30–32, **32**, 33, 67, 69–71, 89, 93, 96
PtlNIStatus 29, 30, **33**, 69–71
PtlPost 92
PtlPut . 53, 57, 62, **63**, 64, 69–71, 76, 78, 99
PtlPutRegion . 57, 62, 64, **64**, 69, 70, 75, 76, 78, 97
PtlSetInvalidHandle 92
PtlTestGetPut 92
PtlTransId 97
PtlWait 98
summary 70

G

gather/scatter . . . . . *see* scatter/gather
generic 92
get . . . . . *see* operations
get ID 37
get uid 36
get_md_handle (field) 67, 78
getput . . . . . *see* operations
Greenberg, David 4

H

Hale, Art 4
handle 28
comparison 68
encoding 28, 35
operations **68**
handle (field) 35
handle1 (field) 68
handle2 (field) 68
handler execution context 57
hardware specific 83
hdr_data (field) 55, 63, 64, 67, 76, 78
header 25
header data 45, 63, 67, 69, 76, 78
header, trusted 25, 36, 37
hint 24
Hoffman, Eric 4
Hudson, Trammell 1, 3

I

I/O vector . . . . . *see* scatter/gather
ID **28**
get 37
job . . . . . *see* job ID
network interface 28
node . . . . . *see* node ID
process . . . . . *see* process ID
thread . . . . . *see* thread ID
uid (get) 36
user . . . . . *see* user ID
id (field) 37
identifier . . . . . *see* ID
iface (field) 32, 40, 41
ignore bits 23, 39, 41
ignore_bits (field) 39, 41
implementation 91
implementation notes 11
implementation, quality 32
inactive 43, 44, 46
indexes, portal 28
initialization **29**
initiator . . *see also* target, [13], 21, 22, 25, 51–54, 62, 64–67, 75–78
initiator (field) 54
interrupt 18, 92
interrupt latency 18
Istrail, Gabi 4

J

jid (field) 38, 54, 61, 99
job ID . . 24, 25, 28, 37, 38, 54, 61, 69, 70, 72, 89
jobid (field) 99
Johnston, Jeanette 4
Jong, Chu 4

K

Kaul, Clint 4

L

LaTeX 98
length (field) 43, 64, 66, 76–78
Levenhagen, Mike 4
limits **31**, 67, 69, 89, 98
link (field) 55
Linux 86
local offset . . . . . *see* offset
local_offset (field) 64, 66, 77
Lustre 12
Lyx 98

M

Maccabe, Arthur B. 1, 3
match bits . 18, 22, 23, 27, 28, 39, 41, 54, 63–67, 69, 76–79
match ID checking 40
match list . . . . . [13], 22, **38**, 39
match list entry . . . . . *see* ME, 38
match_bits (field) 39, 41, 54, 63–67, 76–78
match_id (field) 39–42, 61, 62
matching 40
max_ac_index (field) 31
max_eqs (field) 31
max_getput_md (field) 31, 67
max_interfaces (field) 30
max_md_iovecs (field) 31, 97
max_mds (field) 31
max_me_list (field) 31, 97
max_mes (field) 31
max_offset (field) 94, 96, 97
max_pt_index (field) 31
max_size (field) 44, 46, 55, 72, 97
Maximum length of getput operation 31
McCurley, Kevin 4
MD **42**
    alignment 43, 89
    atomic update 48
    attach 45
    bind 47
    free floating 47
    I/O vector 45
    inactive 43, 44, 46
    message reject 79
    options 44, 53, 97
    pending operation 42, 47, 76
    threshold 49
    truncate 44, 54, 73, 78, 79
    unlink 23, 38, 39, 41–43, 45–51, 53, 55, 69–73, 77, 94–98
    update 48
md (field) 46, 47, 55
md_handle (field) 46–49, 55, 62–66, 76, 77
ME **38**
    attach 39, 40
    ignore bits . . . . . *see* ignore bits
    insert 41
    insert position 38, 69, 72
    match bits . . . . . *see* match bits
    unlink 38, 39, 42, 69–71
me_handle (field) 39, 41, 42, 46
memory descriptor . . . . *see also* MD, [13], 22, **42**
message . . . . . [13]
message operation . . . . . [13]
message rejection 78
messages, receiving **78**
messages, sending **75**
mlength (field) 54
MPI . . . . . [13], 16, 17, 21, 45, 49, 83, 85, 94
    progress rule 16, 18
MPI scalability 16
MPIrecv (func) 83
MPP . . . . . [13]
Myrinet 17, 84, 86

N

NAL . . . . . [13], 84, 91
naming conventions **27**
network . . . . . [13]
network independence 16
network interface *see also* NI, 18, 27–29, **30**, 32, 78, 93, 95
network interface initialization **32**
network interfaces
    multiple 89
network scalability 16
new_md (field) 48, 49
NI distance 34
NI fini 33
NI handle 35
NI init 32
NI status 33
ni_fail_type (field) 53, 55
ni_handle (field) 32–41, 47, 56, 61, 62
nid (field) 37
node . . . . . [13]
node ID 22, 23, 25, 28, 36
NULL MD 43

O

offset 21, 54, 76–78, 95
    local 44, 48, 55, 64–66, 94
    remote 44, 55, 63, 65, 67, 72
offset (field) 55
old_md (field) 48, 49
one-sided operation 17, 22
opening into address space 21

operations
acknowledgment . 33, 44, 50, 51, 75–78, 91, 97
completion 46
get 13, 21, 23, 33, 44, 45, 50, 51, 53, **65**, 66, 70, 72, 75, 77–79, 92
get region **65**
getput 13, 21, 23, 31, 33, 45, 51, 53, **66**, 67, 77–79, 99
one-sided 17, 22
put 13, 17, 19, 21, 23, 33, 44, 45, 50, 51, 53, **62**, 66, 70, 73, 75–79, 92, 96, 97
put region **64**
reply . 19, 21, 33, 42, 44, 45, 51–53, 65, 66, 75, 77, 78
two-sided 17, 22
options (field) 44
OS bypass 16, **18**, 86
Otto, Jim 4

O

P

packetization 60
parallel job 17, 37
Pedretti, Kevin 1, 3
pending operation . . . . . *see* MD
people
Barsis, Ed 4
Barton, Eric 4
Braam, Peter 4
Brightwell, Ron 1, 3
Camp, Bill 4
Fisk, Lee Ann 4
Greenberg, David 4
Hale, Art 4
Hoffman, Eric 4
Hudson, Trammel1 3
Hudson, Trammell 1
Istrail, Gabi 4
Johnston, Jeanette 4
Jong, Chu 4
Kaul, Clint 4
Levenhagen, Mike 4
Maccabe, Arthur B. 1, 3
McCurley, Kevin 4
Otto, Jim 4
Pedretti, Kevin 1, 3
Pundit, Neil 4
Riesen, Rolf 1, 3
Robboy, David 4
Schutt, Jim 4, 91
Sears, Mark 4
Shuler, Lance 4
Stallcup, Mack 4
Underwood, Todd 4
Vigil, Dena 4
Ward, Lee 4
Wheat, Stephen 4
van Dresser, David 4
performance 85
pid (field) 32, 33, 37
portability 30, 83
portal
indexes 28
table 22, 30, 61, 89
table index 38, 41, 61, 73, 76–79
Portals
early versions 12
Version 2.0 12
Version 3.0 12
portals
addressing . . . . . *see* address translation
constants . . . . . *see* constants
constants summary 71
data types **27**, 69
design 85
functions . . . . . *see* functions
functions summary 70
handle 28
multi-threading 26
naming conventions 27
operations . . . . . *see* operations
return codes . . . . . *see* return codes
return codes summary 70
scalability 17
semantics 75
sizes 28
portals3.h 27
position (field) 39, 41
process . . . . . [13], 26, 93
process (field) 34, 35
process aggregation **37**
process ID . 21–25, 28, 32, **36**, 37–40, 63, 67, 69, 96
well known 32, 96
progress 18
progress rule 16, 18, 91
protected space 23, 24
pt_index (field) 39–41, 54, 61–67, 76–78
PTL_AC_INDEX_INVALID (return code) 62, 71, 92
PTL_ACK_REQ (const) 27, 62, 63, 72
PTL_EQ_DROPPED (return code) 57–60, 71
PTL_EQ_EMPTY (return code) . . . 58, 60, 71, 93
PTL_EQ_HANDLER_NONE (const) . . 56, 57, 72
PTL_EQ_INVALID (return code) 46, 47, 49, 57–60, 71, 95
PTL_EQ_NONE (const) 28, 45, 48, 49, 51, 72, 95

PTL_EVENT_ACK (const) 45, 51, 53, 62, 63, 72, 96, 98
P
PTL_EVENT_ACK_END (const) 98
PTL_EVENT_ACK_START (const) 98
PTL_EVENT_GET (const) 96
PTL_EVENT_GET_END (const) . . 50, 53, 65, 72
PTL_EVENT_GET_START (const) 50, 53, 65, 72
PTL_EVENT_GETPUT_END (const) 51, 53, 66, 72
PTL_EVENT_GETPUT_START (const) . . 51, 53, 66, 72
PTL_EVENT_PUT (const) 96
PTL_EVENT_PUT_END (const) . . 50, 53, 62, 72
PTL_EVENT_PUT_START (const) 50, 53, 62, 72
PTL_EVENT_REPLY (const) 96
PTL_EVENT_REPLY_END (const) . . 51, 53, 65, 66, 72
PTL_EVENT_REPLY_START (const) 51–53, 65, 66, 72
PTL_EVENT_SEND_END (const) 51–53, 62, 63, 66, 72, 92
PTL_EVENT_SEND_START (const) . 51, 53, 62, 63, 66, 72, 92
PTL_EVENT_SENT (const) 96
PTL_EVENT_UNLINK (const) . . 42, 45, 47, 51, 53, 55, 72, 97, 98
PTL_FAIL (return code) 30, 68, 71
PTL_HANDLE_INVALID (return code) . . 35, 71
PTL_IFACE_DEFAULT (const) 28, 72, 89
PTL_IFACE_DUP (return code) 97
PTL_IFACE_INVALID (return code) . . 32, 71, 92
PTL_INS_AFTER (const) 39, 41, 72, 98
PTL_INS_BEFORE (const) 39, 41, 72, 98
PTL_INVALID_HANDLE (const) 28, 68, 72
PTL_JID_ANY (const) 29, 61, 72
PTL_JID_NONE (const) 38, 54, 72
PTL_MD_ACK_DISABLE (const) 44, 72
PTL_MD_EVENT_END_DISABLE (const) . . 44, 45, 51, 53, 72, 97, 98
PTL_MD_EVENT_START_DISABLE (const) 44, 45, 51, 53, 72, 97, 98
PTL_MD_ILLEGAL (return code) 46, 47, 49, 64, 66, 71, 92
PTL_MD_IN_USE (return code) . . . 48, 52, 71, 96
PTL_MD_INVALID (return code) 48, 49, 63–67, 71
PTL_MD_IOVEC (const) 44, 45, 72, 97
PTL_MD_MANAGE_REMOTE (const) . . 44, 63, 65, 67, 72, 76, 96, 97
PTL_MD_MAX_SIZE (const) 44, 72, 97, 98
PTL_MD_NO_UPDATE (return code) . 49, 71, 98
PTL_MD_NONE (const) 98
PTL_MD_OP_GET (const) 44, 67, 72, 79
PTL_MD_OP_PUT (const) 44, 67, 73, 79
PTL_MD_THRESH_INF (const) 43, 73
PTL_MD_TRUNCATE (const) 44, 73, 79
PTL_ME_IN_USE (return code) 42, 46, 71
PTL_ME_INVALID (return code) 42, 46, 71
PTL_ME_LIST_TOO_LONG (return code) . . . 40, 42, 71, 96
PTL_NI_INVALID (return code) . . 33–38, 40, 41, 47, 56, 62, 71
PTL_NI_OK (const) 53, 55, 73, 98
PTL_NID_ANY (const) 29, 39, 61, 73
PTL_NO_ACK_REQ (const) . . 62, 63, 73, 75, 76, 98
PTL_NO_INIT (return code) 32–38, 40–42, 46–49, 56–60, 62–67, 71
PTL_NO_SPACE (return code) 32, 33, 40–42, 46, 47, 56, 71
PTL_NO_UPDATE (return code) 98
PTL_OK (return code) 27, 30, 32–38, 40–42, 46–49, 56–60, 62–68, 71
PTL_PID_ANY (const) 29, 32, 39, 61, 73
PTL_PID_INVALID (return code) 33, 71
PTL_PROCESS_INVALID (return code) 35, 40–42, 62–67, 71
PTL_PT_FULL (return code) 41, 71, 97
PTL_PT_INDEX_ANY (const) 61, 73
PTL_PT_INDEX_INVALID (return code) 40, 62, 71
PTL_RETAIN (const) 38, 39, 46, 73
PTL_SEGV (return code) . 29, 30, 33–38, 46, 47, 49, 56, 58–60, 71, 96
PTL_SR_DROP_COUNT (const) . . 29, 34, 73, 89
PTL_SR_INDEX_INVALID (return code) . 34, 71
PTL_TIME_FOREVER (const) 60, 73
PTL_UID_ANY (const) 29, 61, 73
PTL_UNLINK (const) 38, 39, 46, 73, 94
ptl_ac_index_t (type) 28, 69, 76–78
ptl_ack_req_t (type) 62, 69, 72, 73
ptl_eq_handler_t (type) 55, 69, 72
ptl_event_kind_t (type) 50, 69, 72
ptl_event_t (type) 50, 61, 69, 92, 96
ptl_handle_any_t (type) 28, 69, 72
ptl_handle_eq_t (type) 28, 50, 69, 72
ptl_handle_md_t (type) 69, 76–78
ptl_handle_me_t (type) 69
ptl_handle_ni_t (type) 28, 69
ptl_hdr_data_t (type) 69, 76, 78
PTL_ID_ANY (const) 95
ptl_ins_pos_t (type) 38, 69, 72
PTL_INSERT_AFTER (const) 98
PTL_INSERT_BEFORE (const) 98
ptl_interface_t (type) 28, 69, 72
ptl_jid_t (type) 28, 69, 72, 76–78
ptl_match_bits_t (type) 27, 28, 69, 76–78
ptl_md_iovec_t (type) 44, 45, 69, 98

P

ptl_md_t (type) 43, 69
ptl_ni_fail_t (type) 53, 69, 73, 96
ptl_ni_limits_t (type) 31, 69, 98
ptl_nid_t (type) 28, 69, 73, 97
ptl_pid_t (type) 28, 69, 73, 97
ptl_process_id_t (type) . 36, 37, 39, 61, 69, 76–78
ptl_pt_index_t (type) 28, 69, 73, 76–78
ptl_seq_t (type) 69, 96
ptl_size_t (type) 28, 70, 76–78, 92
ptl_sr_index_t (type) 29, 70, 73, 89
ptl_sr_value_t (type) 29, 70
ptl_time_t (type) 70, 73, 98
ptl_uid_t (type) 28, 70, 73, 76–78, 97
ptl_unlink_t (type) 38, 46, 70, 73
PtlACEntry (func) **61**, 69–71, 92
PtlEQAlloc (func) 27, 50, 55, **55**, 57, 69–71
PtlEQCount (func) 93
PtlEQFree (func) 50, **57**, 69–71
PtlEQGet (func) . . . 50, 56–58, **58**, 59–61, 69–71, 92, 93
PtlEQPoll (func) . . 26, 50, 56–59, **59**, 60, 61, 69, 70, 97, 98
PtlEQWait (func) 26, 50, 56–58, **58**, 59–61, 69–71, 92, 93, 98
PtlFini (func) 29, 30, **30**, 70, 71, 93
PtlGet (func) . . . 57, 62, **65**, 66, 69–71, 77, 92, 99
PtlGetId (func) 33, 36, **37**, 38, 69, 70, 96
PtlGetJid (func) **38**, 69, 70
PtlGetPut (func) 31, 57, 62, 66, **67**, 69, 70, 78, 89
PtlGetRegion (func) 57, 62, 65, **66**, 69, 70, 77, 97
PtlGetUid (func) **36**, 70, 98
PtlHandleIsEqual (func) **68**, 69, 70, 92, 98
PtlInit (func) 27, **29**, 70, 71, 93
PtlIsValidHandle (func) 92
PtlMDAlloc (func) 98
PtlMDAttach (func) 42, **45**, 46, 47, 69–71
PtlMDBind (func) 42, **47**, 48, 69–71
PtlMDInsert (func) 94
PtlMDUnlink (func) . . . 43, 47, 48, **48**, 50, 69–71
PtlMDUpdate (func) 43, 46, 48, **48**, 69–71, 92, 95
PtlMEAttach (func) 30, 38, 39, **39**, 40, 41, 69–71, 95, 96
PtlMEAttachAny (func) 30, 38, 40, **40**, 69–71, 97
PtlMEInsert (func) 38, **41**, 69–71, 92
PtlMEUnlink (func) 38, **42**, 69–71
PtlNIBarrier (func) 93
PtlNIDist (func) 31, **34**, 35, 69–71, 89, 92
PtlNIFini (func) 30, **33**, 69–71
PtlNIHandle (func) 31, **35**, 69–71
PtlNIInit (func) 30–32, **32**, 33, 67, 69–71, 89, 93, 96
PtlNIStatus (func) 29, 30, **33**, 69–71
PtlPost (func) 92
PtlPut (func) 53, 57, 62, **63**, 64, 69–71, 76, 78, 99
PtlPutRegion (func) . . . 57, 62, 64, **64**, 69, 70, 75, 76, 78, 97
PtlSetInvalidHandle (func) 92
PtlTestGetPut (func) 92
PtlTransId (func) 97
PtlWait (func) 98
Puma 16
Pundit, Neil 4
purpose **16**
put . . . . . *see* operations
put_md_handle (field) 66, 67, 78

Q

quality implementation 32
quality of implementation 17

R

README 27, 89
receiver-managed 16
Red Storm 91
reliable communication 17, 83
remote offset . . . . . *see* offset
remote_offset (field) 63–67, 76–78, 97
reply . . . . . *see* operations
return codes **29**, 70, 92
PTL_AC_INDEX_INVALID 62, 71, 92
PTL_EQ_DROPPED 57–60, 71
PTL_EQ_EMPTY 58, 60, 71, 93
PTL_EQ_INVALID . . 46, 47, 49, 57–60, 71, 95
PTL_FAIL 30, 68, 71
PTL_HANDLE_INVALID 35, 71
PTL_IFACE_DUP 97
PTL_IFACE_INVALID 32, 71, 92
PTL_MD_ILLEGAL . 46, 47, 49, 64, 66, 71, 92
PTL_MD_IN_USE 48, 52, 71, 96
PTL_MD_INVALID 48, 49, 63–67, 71
PTL_MD_NO_UPDATE 49, 71, 98
PTL_ME_IN_USE 42, 46, 71
PTL_ME_INVALID 42, 46, 71
PTL_ME_LIST_TOO_LONG . 40, 42, 71, 96
PTL_NI_INVALID . . . 33–38, 40, 41, 47, 56, 62, 71
PTL_NO_INIT 32–38, 40–42, 46–49, 56–60, 62–67, 71
PTL_NO_SPACE . 32, 33, 40–42, 46, 47, 56, 71
PTL_NO_UPDATE 98
PTL_OK 27, 30, 32–38, 40–42, 46–49, 56–60, 62–68, 71
PTL_PID_INVALID 33, 71
PTL_PROCESS_INVALID 35, 40–42, 62–67, 71

R
PTL_PT_FULL 41, 71, 97
PTL_PT_INDEX_INVALID 40, 62, 71
PTL_SEGV . . 29, 30, 33–38, 46, 47, 49, 56, 58–60, 71, 96
PTL_SR_INDEX_INVALID 34, 71
summary 70
Riesen, Rolf 1, 3
rlength (field) 54
RMPP . . . . . [13], 17
Robboy, David 4

S

scalability **17**, 83, 85
guarantee 17
MPI 16
network 16
scatter/gather 44, 45, 69, 72, 98
Schutt, Jim 4, 91
Sears, Mark 4
semantics 75
send 21, 51, 62
send event 51, 53, 62, 66, 72
sequence (field) 55
sequence number 55, 69
Shuler, Lance 4
size (field) 60
sizes 28
sockets 83, 86
space
application 23
protected 23
split event sequence . . . . . *see* event start/end
Stallcup, Mack 4
start (field) 43
state 17, 83
status (field) 34
status registers 29, 89
status_register (field) 34
structure fields and argument names
ac_index 61–67, 76–78
ack_req 63, 64
actual 32, 33
base 41, 42
count 56
desired 32
distance 34, 35
eq_handle 45–49, 56–59
eq_handler 56, 57
eq_handles 60
event 58–60
get_md_handle 67, 78
handle 35
handle1 68
handle2 68
hdr_data 55, 63, 64, 67, 76, 78
id 37
iface 32, 40, 41
ignore_bits 39, 41
initiator 54
jid 38, 54, 61, 99
jobid 99
length 43, 64, 66, 76–78
link 55
local_offset 64, 66, 77
match_bits 39, 41, 54, 63–67, 76–78
match_id 39–42, 61, 62
max_ac_index 31
max_eqs 31
max_getput_md 31, 67
max_interfaces 30
max_md_iovecs 31, 97
max_mds 31
max_me_list 31, 97
max_mes 31
max_offset 94, 96, 97
max_pt_index 31
max_size 44, 46, 55, 72, 97
md 46, 47, 55
md_handle 46–49, 55, 62–66, 76, 77
me_handle 39, 41, 42, 46
mlength 54
new_md 48, 49
ni_fail_type 53, 55
ni_handle 32–41, 47, 56, 61, 62
nid 37
offset 55
old_md 48, 49
options 44
pid 32, 33, 37
position 39, 41
process 34, 35
pt_index 39–41, 54, 61–67, 76–78
put_md_handle 66, 67, 78
remote_offset 63–67, 76–78, 97
rlength 54
sequence 55
size 60
start 43
status 34
status_register 34
target_id 63–67, 76–78
threshold 43
timeout 60
type 54
uid 36, 54, 61
unlink 94
unlink_nofit 96
unlink_op 39, 41, 46, 47, 96

S
user_ptr 45
which 59, 60
summary **68**
SUNMOS . . . . . [13], 16
swap 21, 31, 66, 70

T

target . . *see also* initiator, 13, [14], 17, 18, 21, 22, 36, 51–53, 62, 63, 65–67, 75–78
target_id (field) 63–67, 76–78
TCP/IP 16, 83, 86, 91
thread . . . . . [14], 26, 93, 97
thread ID 36
threshold 23, 43, 46, 49, 55, 94, 95
threshold (field) 43
timeout 59
timeout (field) 60
truncate 44, 54, 73, 78, 79
trusted header 25, 36
two-sided operation 17, 22
type (field) 54
types . . . . . *see* data types
ptl_ac_index_t 28, 69, 76–78
ptl_ack_req_t 62, 69, 72, 73
ptl_eq_handler_t 55, 69, 72
ptl_event_kind_t 50, 69, 72
ptl_event_t 50, 61, 69, 92, 96
ptl_handle_any_t 28, 69, 72
ptl_handle_eq_t 28, 50, 69, 72
ptl_handle_md_t 69, 76–78
ptl_handle_me_t 69
ptl_handle_ni_t 28, 69
ptl_hdr_data_t 69, 76, 78
ptl_ins_pos_t 38, 69, 72
ptl_interface_t 28, 69, 72
ptl_jid_t 28, 69, 72, 76–78
ptl_match_bits_t 27, 28, 69, 76–78
ptl_md_iovec_t 44, 45, 69, 98
ptl_md_t 43, 69
ptl_ni_fail_t 53, 69, 73, 96
ptl_ni_limits_t 31, 69, 98
ptl_nid_t 28, 69, 73, 97
ptl_pid_t 28, 69, 73, 97
ptl_process_id_t . . . 36, 37, 39, 61, 69, 76–78
ptl_pt_index_t 28, 69, 73, 76–78
ptl_seq_t 69, 96
ptl_size_t 28, 70, 76–78, 92
ptl_sr_index_t 29, 70, 73, 89
ptl_sr_value_t 29, 70
ptl_time_t 70, 73, 98
ptl_uid_t 28, 70, 73, 76–78, 97
ptl_unlink_t 38, 46, 70, 73

U

uid (field) 36, 54, 61
undefined behavior 29, 30, 33, 46, 57
Underwood, Todd 4
unexpected messages 16
unlink
MD . . . . . *see* MD
ME . . . . . *see* ME
unlink (field) 94
unlink_nofit (field) 96
unlink_op (field) 39, 41, 46, 47, 96
unreliable networks 60, 84
update . . . . . *see* PtlMDUpdate
user data 45
user ID 24, 25, 28, **36**, 54, 70, 73, 95, 96
user memory 19
user space 17
user-level bypass . . . . . *see* application bypass
user_ptr (field) 45

V

van Dresser, David 4
VIA . . . . . [14]
Vigil, Dena 4

W

Ward, Lee 4
web site 91
Wheat, Stephen 4
which (field) 59, 60
wire protocol 17, 21, 75, 83

Z

zero copy **18**
zero-length buffer 43

(*n*) page *n* is in the bibliography.
[*n*] page *n* is in the glossary.
n page of a definition or a main entry.
n other pages where an entry is mentioned.

DISTRIBUTION:

1 Arthur B. Maccabe
University of New Mexico
Department of Computer Science
Albuquerque, NM 87131-1386

1 Trammell Hudson
c/o OS Research
1527 16th NW #5
Washington, DC 20036

1 Eric Barton
9 York Gardens
Clifton
Bristol BS8 4LL
United Kingdom

1 MS 0806
Jim Schutt, 4336

1 MS 0817
Doug Doerfler, 1422

1 MS 0817
Sue Kelly, 1422

1 MS 1110
Ron Brightwell, 1423

1 MS 1110
Neil Pundit, 1423

4 MS 1110
Rolf Riesen, 1423

1 MS 1110
Lee Ward, 1423

1 MS 1110
Ron Oldfield, 1423

1 MS 1110
Kevin Pedretti, 1423

1 MS 1110
Keith Underwood, 1423

2 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 4536

Sandia National Laboratories